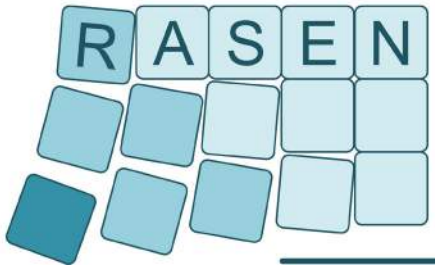


Compositional Risk Assessment and Security Testing of Networked Systems

Deliverable D4.2.2

Techniques for Compositional Risk-Based Security Testing v.2

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:	D4.2.2
Nature of deliverable:	Report
Dissemination level:	PU
Internal version number:	1.0
Contractual delivery date:	2014-09-30
Actual delivery date:	2015-09-30
Responsible partner:	Fraunhofer

Contributors

Editor(s)	Martin Schneider (FOKUS)
Contributor(s)	Fredrik Seehusen (SINTEF), Fabien Peureux (SMA), Julien Botella (SMA), Martin Schneider (FOKUS), Johannes Viehmann (FOKUS)
Quality assuror(s)	SAG, SMA

Version history

Version	Date	Description
0.1	20-06-06	TOC proposition
0.2	27-08-14	SINTEF contribution
0.3	10-09-14	SMA contribution
0.4	11-09-14	FOKUS contribution
0.5	19-09-14	SINTEF contribution updated based on internal review feedback
0.6	19-09-14	FOKUS and SMA contribution updated based on internal review feedback
1.0	29-09-14	Final quality check

Abstract

Work package 4 develops 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 networked systems. This report provides the evolved results based on the previous deliverable D4.2.1. The results comprise risk-based testing using CAPEC attack patterns. An improved security test pattern-based approach for test case generation is presented as well as improvements of the behavioral fuzzing approach in order to address certain vulnerabilities. Test case generation using security test patterns together with a test purpose language is extended for security testing. In addition, first results regarding security testing metrics are described. This deliverable will be refined by D4.2.3.

Keywords

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

Executive Summary

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

This document provides techniques for test procedure identification, prioritization and selection and test case derivation based on risk assessment results. The starting point for the development of these techniques is defined by the RASEN deliverable D4.2.1 that presented the first results on these tasks.

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 second of three 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 and selection of security test cases based on risk assessment result, and deriving security test cases from risk assessment results.

Additionally to tasks addressed by D4.2.1, this deliverable covers also metric aspects regarding the research question

What are suitable metrics for quantitative security assessment in complex environments?

First results regarding this research questions are a categorization of security testing metrics and exemplary definitions of security testing metrics.

Table of contents

1 Introduction

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 by means of reasonable risk coverage and probability metrics.

This document provides techniques for test procedure identification, prioritization and selection and test case derivation based on risk assessment results. The starting point for the development of these techniques is defined by the RASEN deliverable D4.2.1 that presented the first results on these tasks.

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 second one 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 and selection of security test cases based on risk assessment result, and deriving security test cases from risk assessment results.

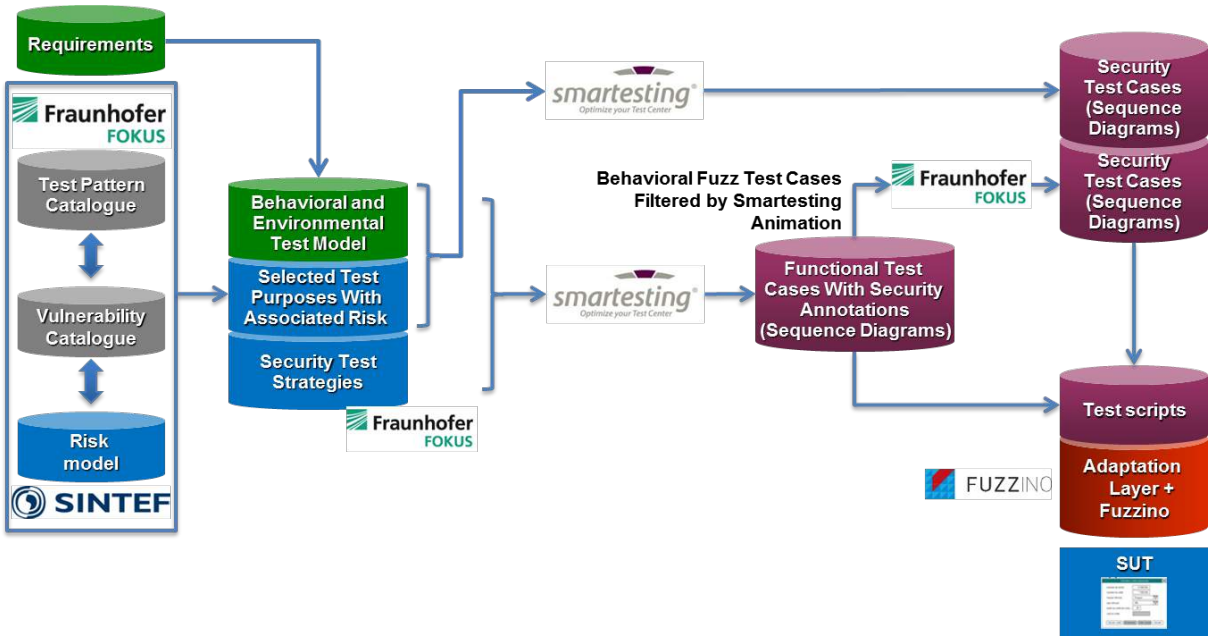


Figure 1 – Overall process of security testing based on risk assessment results

An approach for risk-based test procedure identification, prioritization and selection based on attack patterns from the CAPEC database is presented in Section 2. This constitutes the starting point for the risk-based testing process. Refinements of security test patterns are described in Section 3 comprising security test strategies. Section 4 describes test pattern instantiation and test sequence generation based on security test purposes and Section 5 security testing techniques for selected OWASP Top 10 vulnerabilities employing behavioral fuzzing techniques.

Additionally to D4.2.1, this deliverable covers also metric aspects regarding the research question

What are suitable metrics for quantitative security assessment in complex environments?

First results regarding this research questions are a categorization of security testing metrics and first, exemplary definitions of security testing metrics and is covered by Section 6.

2 Using CAPEC for Risk-Based Testing

CAPEC is a catalogue of common security attacks. In this Section, we describe a technique for using CAPEC for risk-based test procedure identification, prioritization and selection. As illustrated in Figure 2, the techniques are meant to be used as part of a three step process where in step I, the CAPEC catalog is automatically transformed into a generic risk model. In step 2, the generic risk model is manually refined to make it specific to the target of evaluation. We discuss some techniques for doing this. In step 3, the target specific risk model obtained from step 2 is used as a basis for test procedure identification, prioritization and selection. The outcome of step 3 is a prioritized list of selected test procedures that are meant to be used as a starting point for test implementation and execution.

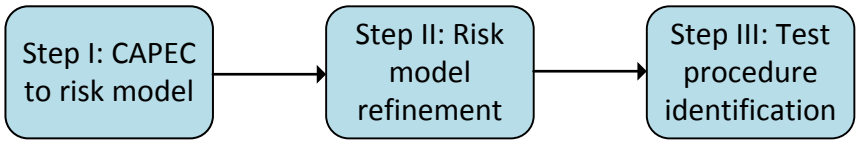


Figure 2 – The steps of the process

2.1 Step I: From CAPEC to Generic CORAS Risk Models

In this Section, we describe a technique for automatically generating a risk model from the CAPEC catalog. The kind of risk model we use are so-called CORAS threat diagrams. We first describe CORAS risk models (in Section 2.1.1), and the CAPEC catalog (in Section 2.1.2), before describing the translation from CAPEC to the CORAS risk model (in Section 2.1.3).

2.1.1 CORAS Risk Models

In this Section, we describe CORAS risk models which are used to document risks and events/circumstances that can cause risks. As illustrated by the example in Figure 3, a CORAS risk model is a directed acyclic graph where every node is of one of the following kinds:

- **Threat** A potential cause of an unwanted incident or threat scenario.
- **Threat scenario** A chain or series of events that is initiated by a threat and that may lead to an unwanted incident.
- **Unwanted incident** An event that harms or reduces the value of an asset.
- **Asset** Something to which a party assigns value and hence for which the party requires protection.

Note that risks can also be represented in a CORAS risk model, but these correspond to pairs of unwanted incidents and assets. If an unwanted incident harms exactly one asset, as is the case in Figure 3, then this unwanted incident will represent a single risk.

A relation in a CORAS model may be of one of the following kinds:

- **Initiates relation** going from a threat A to a threat scenario or unwanted incident B, meaning that A initiates B.
- **Leads to relation** going from a threat scenario or unwanted incident A to a threat scenario or un-wanted incident B, meaning that A leads to B.
- **Harms relation** going from an unwanted incident A to an asset B, meaning that A harms B.

Relations and nodes may have assignments, in particular

- **Likelihood values** may be assigned to a threat scenario and unwanted incident A, estimating the likelihood of A occurring.
- **Conditional likelihood values** may be assigned leads to relations going from A to B, estimating the conditional likelihood that B occurs given that A has occurred.

- **Consequence values** may be assigned to harms relations going from A to B, estimation the consequence the occurrence of A has on B.
- **Vulnerabilities** may be assigned to leads to relations going from A to B, describing a weakness, flaw or deficiency that opens for A leading to B.

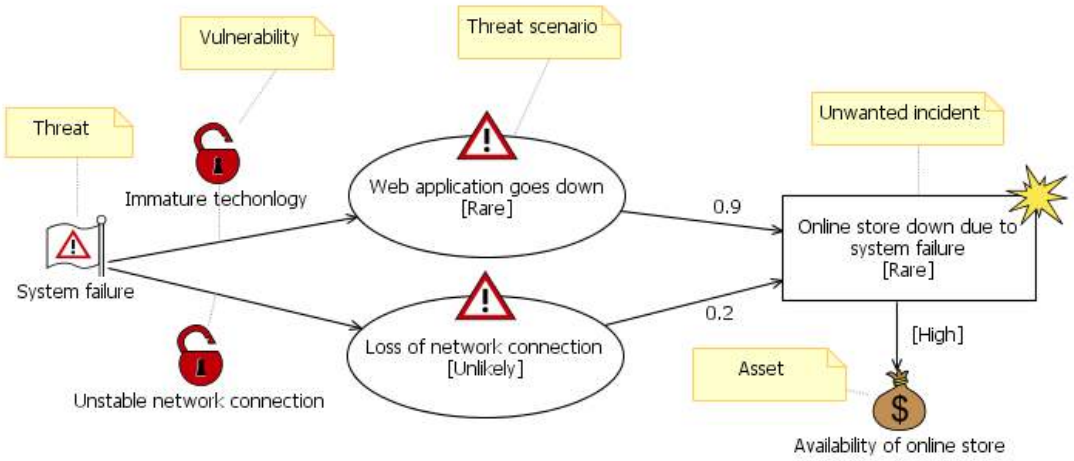


Figure 3 – Example of a CORAS risk model

2.1.2 Common Attack Pattern Enumeration and Classification (CAPEC)

CAPEC is catalogue containing common security attack patterns. CAPEC provides a template/data format for documenting common characteristics of security attacks. The template is shown in Table 1 below.

Name	The Name is a descriptive name used to give the reader an idea of the meaning behind the compound attack pattern structure.
Typical severity	This element reflects the typical severity of an attack on a scale of {Very Low, Low, Medium, High, Very High}. USAGE: This element is used to capture an overall typical average value for this type of attack with the understanding that it will not be completely accurate for all attacks.
Description	This field provides a description of this Structure, whether it is an Attack Pattern, Category or Compound Element. Its primary subelement is Description_Summary which is intended to serve as a minimalistic description which provides the information necessary to understand the primary focus of this entry. Additionally, it has the subelement Extended_Description which is optional and is used to provide further information pertaining to this attack pattern.
Attack prerequisites	This field describes the conditions that must exist or the functionality and characteristics that the target software must have or behavior it must exhibit for an attack of this type to succeed
Typical likelihood of exploit	This element represents the typical likelihood that the attack will succeed, and provides a likelihood estimate and an explanation that qualifies the estimate. USAGE: This element is used to capture an overall typical average value for this type of attack with the understanding that it will not be completely accurate for all attacks.
Methods of attack	This field describes the mechanism of attack used by this pattern. This field can help define the applicable attack surface required for this attack.
Examples-instances	This element represents a container of one or more example instances. An example instance details an explanatory example or demonstrative exploit instance of this attack, USAGE: This element is

	used to help the reader understand the nature, context and variability of the attack in more practical and concrete terms.
Attacker skill or knowledge required	This element reflects the level of knowledge or skill required to execute this type of attack on a scale of { Low, Medium, High }. USAGE: This element is used to represent the level with respect to a specified type of skill or knowledge, e.g., low - basic SQL knowledge, high - expert knowledge of LINUX kernel, etc.
Resources required	This field describes the resources (CPU cycles, IP addresses, tools, etc.) required by an attacker to effectively execute this type of attack.
Probing techniques	A probing technique describes a method used to probe and reconnoiter a potential target to determine vulnerability and/or to prepare for this type of attack.
Indicator-warnings of attack	This element provides an explanatory description of the indicator warning of attack.
Solutions and mitigations	This element represents a container of one or more solutions or mitigations. A solution or mitigation describes actions or approaches to prevent or mitigate the risk of this attack by improving the resilience of the target system, reduce its attack surface or to reduce the impact of the attack if it is successful.
Attack motivation-consequences	This element represents a container of one ore more attack motivation consequences. Attack motivation consequence represents the desired technical results that could be achieved/leveraged by this attack pattern, represented as an enumerated list of defined adversary motivations/consequences. USAGE: This element is used to identify specific technical results that could be leveraged to achieve the adversary's business or mission objective. This information is useful for aligning attack patterns to threat models and for determining which attack patterns are relevant for a given context.
Technical context	This element characterizes the technical context where this pattern is applicable
Injection vector	This element details the mechanism and format of an input-driven attack of this type. Injection vectors take into account the grammar of an attack, the syntax accepted by the system, the position of various fields, and the ranges of data that are acceptable.
Payload	This element describes the code, configuration or other data to be executed or otherwise activated as part of an injection-based attack of this type.
Activation zone	This element describes the area within the target software that is capable of executing or otherwise activating the payload of an injection-based attack of this type. The activation zone is where the intent of the attacker is put into action. The activation zone may be a command interpreter, some active machine code in a buffer, a client browser, a system API call, etc.
Payload activation impact	This element provides an explanatory description of the payload activation impact.
CIA impact	This element characterizes the typical relative impact of this pattern on the confidentiality, integrity, and availability of the targeted software.
CWE ID (Related weaknesses)	The CWE_ID is a field that exists for all weaknesses enumerated in the Common Weakness Enumeration (CWE). It is a unique value that allows each weakness to be unambiguously identified. The CWE_ID field for the attack pattern contains the value of the CWE_ID for the specific related weakness.
Relevant security requirements	This element represents a container of one or more relevant security requirements. A relevant security requirement is a general security requirement that is relevant to this type of attack.
Related security principles	This element represents a container of one or more related security principles. A principle is defined as a rule or standard for good behavior. A related security principle is a security rule or practice that

	impedes this attack pattern. USAGE: Usage defined in NIST SP 800-27A, "Engineering Principles for Information Technology Security", Revision A.
Related guidelines	This element represents a container of one or more related guidelines. A related guideline represents a security guideline that is relevant to identifying or mitigating this type of attack. USAGE: It would be helpful to provide a usage reference. However links to security principle and guideline documentation on the BSI site appear to be broken. NIST SP 800-27 uses the terms principle and guideline interchangeably.
References	The References element contains one or more Reference elements, each of which provide further reading and insight into this attack pattern.

Table 1– CAPEC template

Our goal is to automatically transform a set of CAPEC attack patterns into a CORAS risk model. To determine what parts of the CAPEC pattern that are relevant for translation into a risk model, we must first ask what can be represented in the CORAS risk model. In answer to this question, we believe that the following information about a security attack can be represented and would be of value in a risk model:

- A: The name of the attack.
- B: An estimate of how likely it is that the attack is initiated.
- C: An estimate of how likely it is that the attack will succeed given that it is initiated.
- D: A list of consequences/unwanted incidents which a successful attack can cause/lead to.
- E: An estimate of how likely it is that a successful attack will lead to the unwanted incidents.
- F: A list of assets that can be affected by the unwanted incidents of successful attacks.
- G: A description of which assets that can be harmed by an unwanted incident.
- H: An estimate of the consequence that an unwanted incident has on each of its assets.
- I: A list of vulnerabilities that can be exploited by the attack.

By examining the CAPEC template, we see that the items that can be derived from a CAPEC pattern are A, C, D, F, G, and H. These items can be derived from the attributes: Name, Typical likelihood of exploit, Attack motivation-consequences, CIA impact, and CWE ID (Related weaknesses). For the purpose of defining the translation from CAPEC to risk models, we will refer to a CAPEC patterns with only the above mentioned attributes described as specified in Table 2 as a *CAPEC instance*.

Name	A pair *(ID,N)* where *ID* denotes the identifier of the attack pattern and *N* denotes the name of the pattern.
Typical likelihood of exploit	A likelihood *LE* denoting the likelihood that the attack will succeed.
Attack motivation-consequences	A list *(TI1, S1), (TI2, S2), …, (TIn, Sn)* of *n* pairs of the form *(TI,S)*, where *TI* denotes the name of a technical impact and *S* denotes the scope of *TI* given as a subset of the set {Availability, Confidentiality, Integrity}
CIA impact	A triple *(cia_c, cia_i, cia_a)* denoting the impact/consequence the attack has on confidentiality, integrity, and availability, respectively.
CWE ID (Related weaknesses)	A list *v1, v2, …,vn* of *n* elements denoting CWE vulnerabilities that can be exploited by the attack.

Table 2– CAPEC instance

In Table 3 and Table 4 we show two examples of CAPEC instances which describe the CAPEC patterns 34 and 62, respectively.

Name	(CAPEC-34, HTTP Response Splitting)
Typical likelihood of exploit	Medium
Attack motivation-consequences	(Execute unauthorized code or commands, {Confidentiality, Integrity, Availability}),

	(Gain privileges / assume identify, {Confidentiality})
CIA impact	(High, High, Low)
CWE ID (Related weaknesses)	CWE-113 Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting'), CWE-697 Insufficient Comparison, CWE-707 Improper Enforcement of Message or Data Structure, CWE-713 OWASP Top Ten 2007 Category A2 - Injection Flaws

Table 3– CAPEC-34 instance example

Name	(CAPEC-62,Cross Site Request Forgery (aka Session Riding))
Typical likelihood of exploit	High
Attack motivation-consequences	(Read application data, {Confidentiality}), (Modify application data, {Integrity}), (Gain privileges / assume identity, {Confidentiality})
CIA impact	(High, High, Low)
CWE ID (Related weaknesses)	CWE-352 Cross-Site Request Forgery (CSRF), CWE-664 Improper Control of a Resource Through its Lifetime, CWE-732 Incorrect Permission Assignment for Critical Resource, CWE-716 OWASP Top Ten 2007 Category A5 - Cross Site Request Forgery (CSRF)

Table 4– CAPEC-62 instance example

2.1.3 From CAPEC Instances to Generic CORAS Risk Models

Having explained what is meant by a CORAS risk model and a CAPEC instance we now describe the translation from CAPEC to CORAS risk model. Because there is information that we would like to represent in a risk model, but which cannot be derived from a CAPEC pattern, we need to supply this additional information as input to the transformation. In particular, we need, in addition to the CAPEC instance,

- a mapping *lm* from CAPEC likelihoods to CORAS likelihoods denoted*;*
- a default initiation likelihood *dil,* specifying the likelihood that an attack will be initiated*;*
- a default technical impact likelihood *dtil* specifying the conditional likelihood of a successful attack leading to a technical impact

Given this information, our transformation to a CORAS risk model will in the general case result in the risk model shown in Figure 4

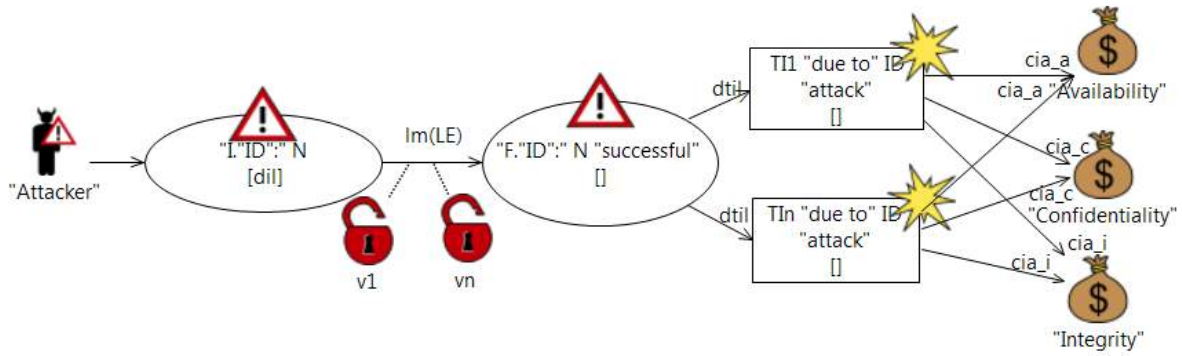


Figure 4– CORAS risk model showing outcome of translation function

To distinguish between variables and strings/constants, we have in Figure 4 denoted all non-variables inside quotation marks. For instance, we have written "Attacker", meaning that Attacker is not a

variable, but should appear as a string. The variables in the diagram such as *ID*, *N*, *v1*,*dil*, etc. are all taken from the CAPEC instance which is assumed to be the input to the translation (see Table 2).

As illustrated in Figure 4, each CAPEC instance is translated into two threat scenarios: one threat scenario corresponding to the initiation of the attack, and one threat scenario corresponding to a *successful* attack. The threat scenario describing attack initiation is given likelihood *dil*. The condition likelihood that the attack will be successful given that it is initiated is given by *lm(LE),* i.e. the exploit likelihood of the CAPEC instance *LE* translated to the CORAS model likelihood by function *lm*.

Given that the attack described by the CAPEC instance is successful, it can lead to one or more technical impacts with conditional likelihood *dtil*. Each technical impact of the CAPEC instance is translated to an unwanted incident in the CORAS model. These unwanted incidents may in turn be connected to one of the three assets, Availability, Confidentiality, or Integrity, and the consequences of the unwanted incidents towards these is given by the CIA values of the CAPEC instance.

The assets that a technical impact is connected to are decided by the scope of the technical impact. For instance, if the scope of the technical impact includes all three assets, then the technical impact will be connected to all the three assets. If the scope only includes e.g. Confidentiality, then the technical impact will only be connected to the Confidentiality asset.

As an example, assume that we supply the following input to the translation:

- The CAPEC instance example shown in Table 3.
- A mapping *lm* defined by the mapping {Low ->eLow, Medium ->eMedium, High ->eHigh}
- A default initiation likelihood iHigh
- A default technical impact likelihood tMedium

then the output of the translation will be the CORAS risk model shown in Figure 5.

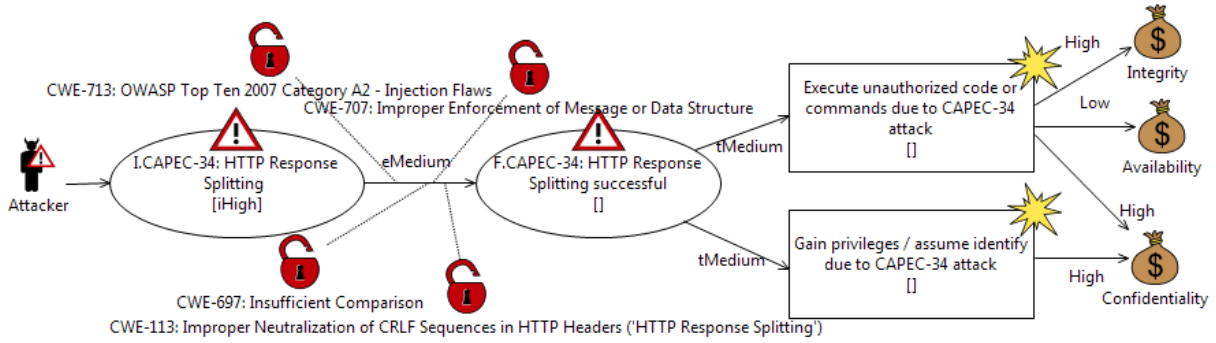


Figure 5– Result obtained by applying the translation to the CAPEC instance example

2.2 Step II: From Generic CORAS Risk Models to Target Specific Risk Models

The translation of CAPEC instances results in a CORAS risk model which is not specific to a particular target of evaluation. For this reason, we suggest that CORAS risk model be manually refined to make it more relevant for a particular target of evaluation. There are several different ways that the CORAS risk model can be refined. In this Section, we cover the most important ones.

2.2.1 Refinement of Likelihood and Consequence Values

All likelihood and consequences of the CORAS risk model obtained from a set of CAPEC instances are not specific to the target of evaluation. One way of making the risk model more specific to the target of evaluation is therefore to examine each likelihood and consequence estimate of the risk model, and adjust them as necessary.

An example of this kind of refinement is shown in Figure 6, where we see that many of the likelihood values of the risk model in Figure 5 have been refined.

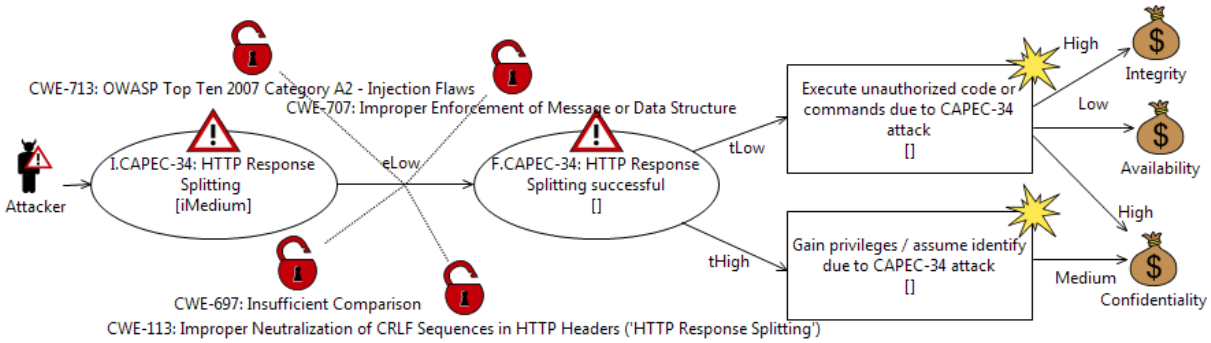


Figure 6 – Example of refinement of likelihood and consequence values

2.2.2 Refinement by Element Splitting

In some cases, it may be that some of the attacks or technical impacts derived from the CAPEC instances are described in a too generic way. In these cases, it might be relevant to refine the risk model by splitting threat scenarios or unwanted incidents. An example is shown in Figure 7, where we assume that it is of interest to distinguish between different features of the target of evaluation that are subject to the attack. The risk model of Figure 7 can be seen as a refinement (by splitting) of Figure 5 where the threat scenario *I.CAPEC-34: HTTP Response Splitting* has been split into two threat scenarios.

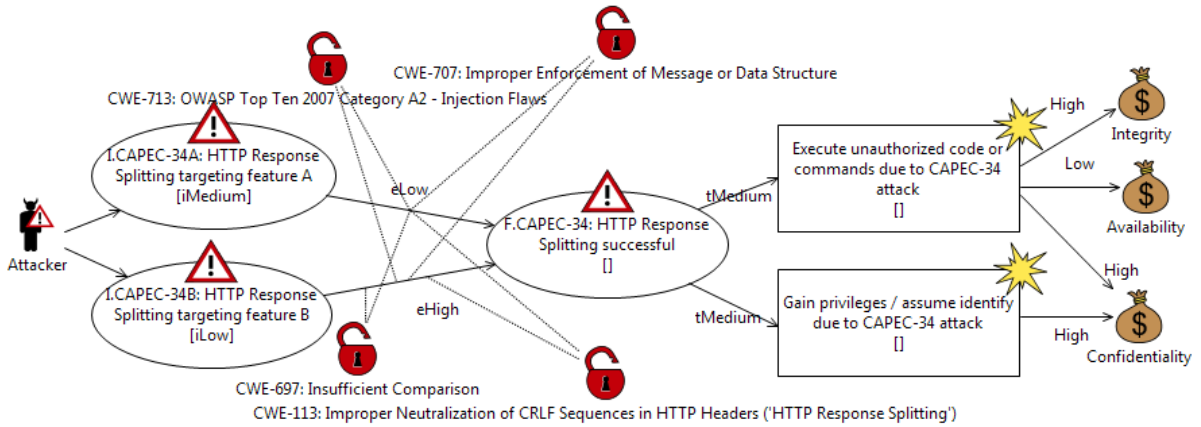


Figure 7– Example of refinement by element splitting

2.2.3 Refinement by Element Merging

In certain cases, it might be relevant to merge threat scenarios or unwanted incidents if they describe similar phenomena. In particular, this may be relevant for the unwanted incidents. If many CAPEC instances are transformed into a risk model, then we can potentially end up with a great number of possible risks (recall that a risk corresponds to an unwanted incident that harms an asset). Therefore merging similar unwanted incidents might be a good way of making the risk model more understandable.

As an example, consider the risk model shown in Figure 8, which has been obtained by translating the CAPEC instances of Table 3 and Table 4. In the risk model, we can see two unwanted incidents with a similar effect (Gain privileges / assume identify). In Figure 9, these unwanted incidents have been merged into one unwanted incident.

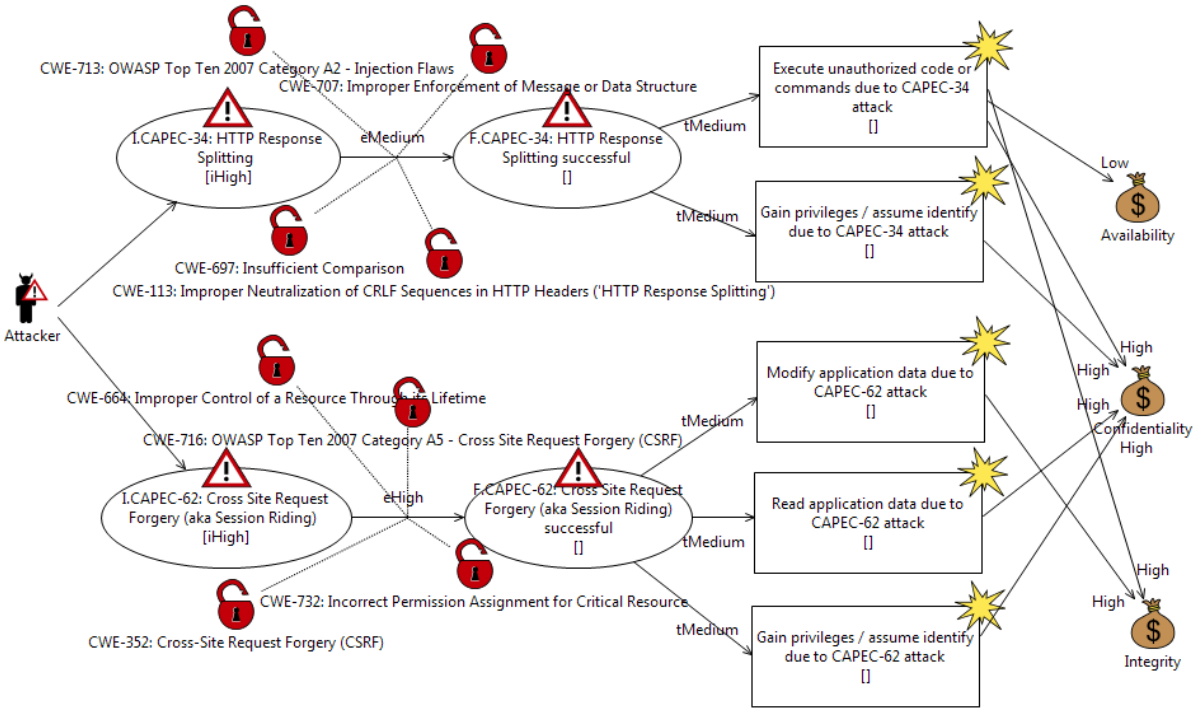


Figure 8– Risk model obtained by translation of CAPEC-34 and CAPEC-62

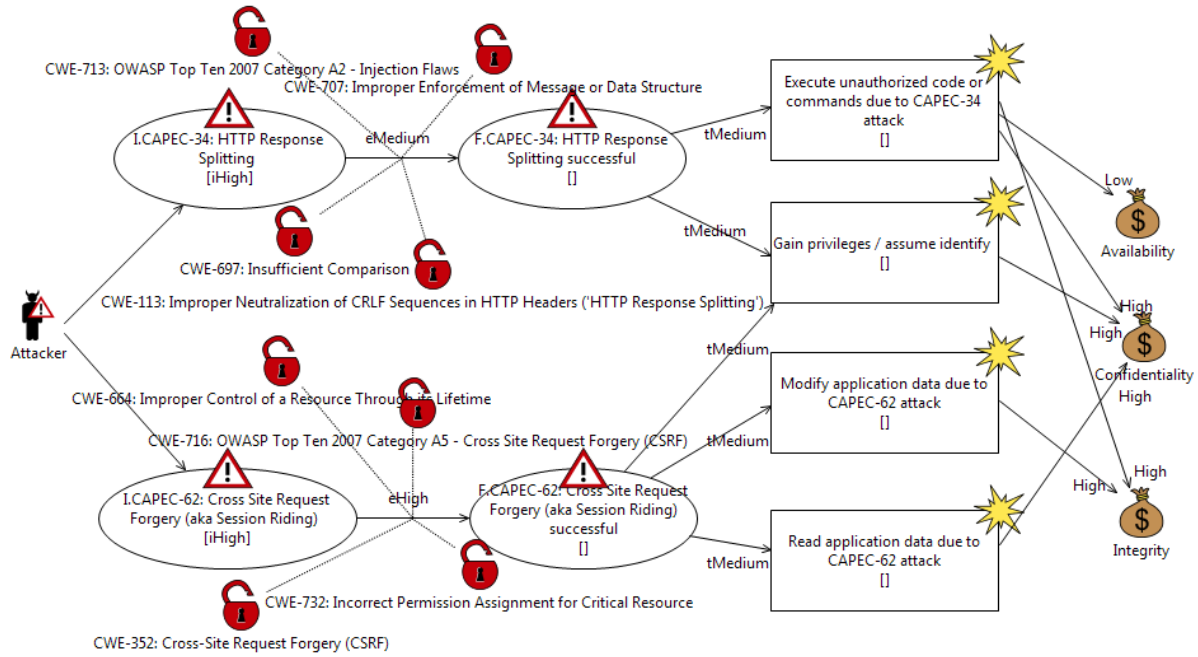


Figure 9 – Example of refinement by element merging

2.2.4 Refinement by Element Addition

The final kind of refinement that we will consider is refinement by element addition. This kind of refinement may be particularly relevant for defining new risks that are specific to the target of evaluation. After all, all unwanted incidents in a risk model derived from CAPEC instances correspond to technical impacts which are described in a quite general manner. Defining new unwanted incidents

which are more specific to the target of evaluation and that can be caused by the technical impacts may therefore be relevant.

An example of this is given in Figure 10 where the risk model of Figure 8 has been refined by adding three new unwanted incidents to the risk model, and connecting these to the old unwanted incidents and the assets.

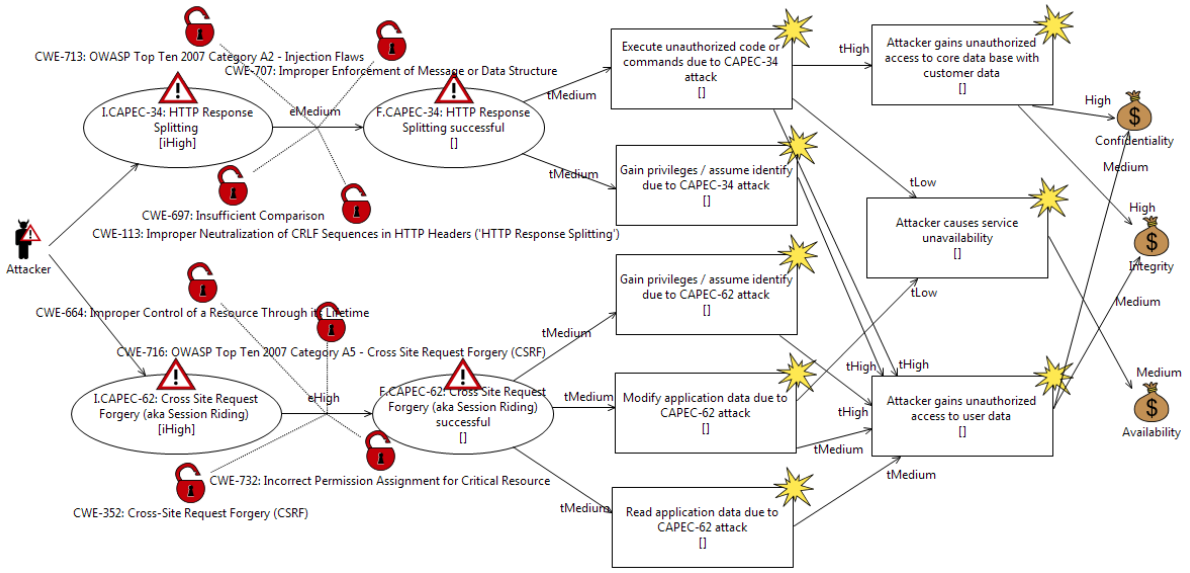


Figure 10– Example of refinement by element addition

2.3 Step III: From Specific Risk Models to Test Procedures

In this Section, we describe how test procedures can be derived from the risk models.

2.3.1 Risk Evaluation and Visualization

Given that we have defined all likelihood values precisely and that we have a risk model that is complete in the sense that all initial threat scenarios and all edges/transitions have been given likelihood values, we can use the calculation technique described in [1] and [2] to calculate the likelihood of all nodes in a risk model. In particular, we can calculate the likelihood of the risks, i.e. the unwanted incidents that have an impact on the assets that we are interested in.

To illustrate this, assume that we have the following likelihood scales:

Likelihood	Definition	Interval
iLow	0-1 times per year	[0, 1>:1y
iMedium	1-30 times per year	[1,30>:1y
iHigh	Over 30 times per year	[30,Infinity>:1y

Table 5 – Likelihood scale for estimating risk and attack initiation

Likelihood	Definition	Probability
eLow	1 out of 100000 attacks successful – 1 out of 10000 successful	[0.00001 – 0.0001>
eMedium	1 out of 10000 attacks successful – 1 out of 1000 successful	[0.0001 – 0.001>
eHigh	1 out of 1000 attacks is successful – 1 out 10 attacks is successful	[0.001 – 0.1>

Table 6 – Likelihood scale for estimating probability of successful attacks

Likelihood	Definition	Probability
tLow	0 out of 10 successful attacks will cause an incident - 1 out of 10 successful attacks will cause an incident	[0.0 - 0.10>
tMedium	1 out of 10 successful attacks will cause an incident - 1 out of 4 successful attacks will cause an incident	[0.10 - 0.25>
tHigh	1 out of 4 successful attacks will cause an incident - 1 out of 2 attacks will cause an incident	[0.25 - 0.5>

Table 7 – Likelihood scale for estimating the probability that a successful attack will cause an unwanted incident

Then the result of calculating all the likelihood values of the risk model given in Figure 10 is shown in Figure 11.

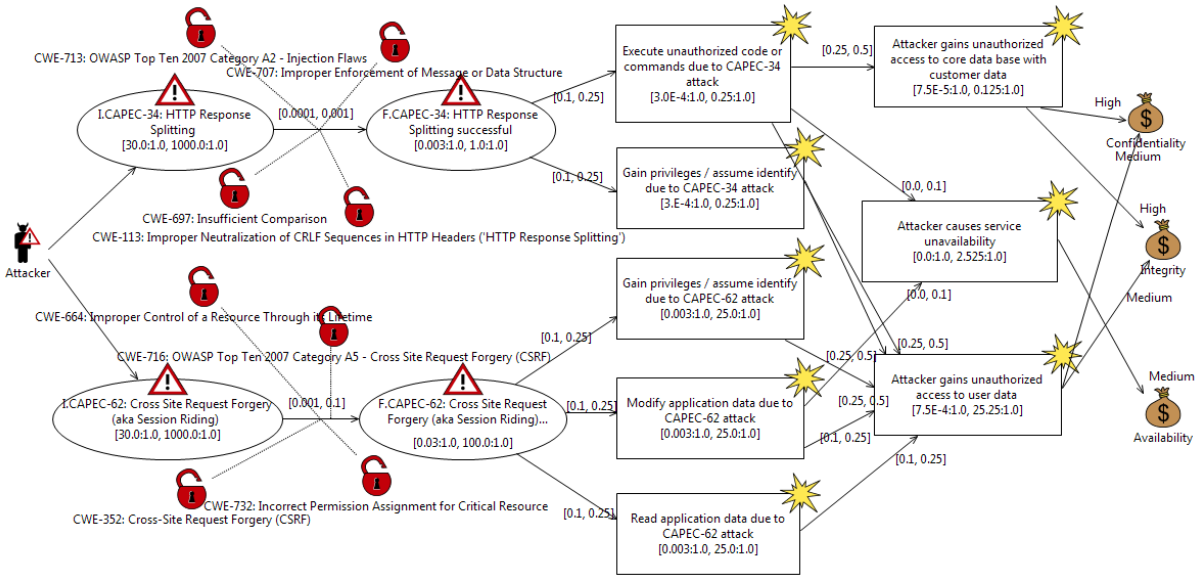
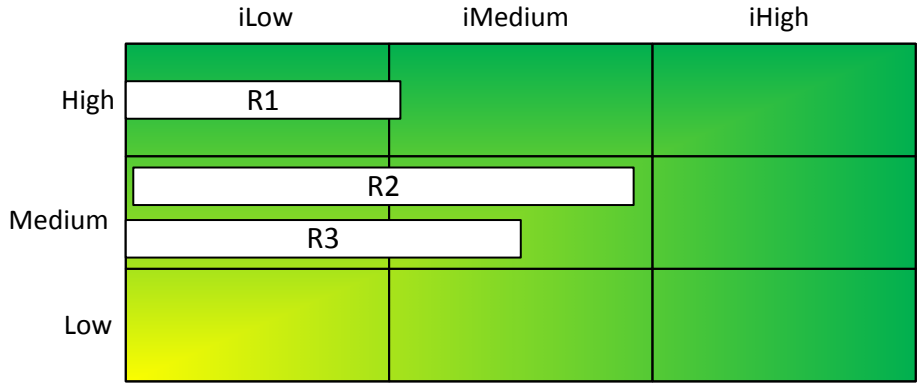


Figure 11 – Risk model with calculated likelihood values

In Figure 11, the risks are the three unwanted incidents on the far right hand side that are associated with the assets. These risks can also be visualized in a risk matrix. This is a common way of risk visualization. However, if some measure of likelihood confidence is used, then we strongly suggest that should be represented visually in the risk matrix. For example, if intervals are used for likelihood estimates and the width of the intervals are used as a confidence measure, then the risk should be represented in the matrix as a box spanning the likelihood axis such that the interval is visualized.

In Figure 12, we have visualized the three risks of Figure 11 in a risk matrix. Here the vertical axis shows the consequence scale and the horizontal axis shows the likelihood scale. Since the likelihoods of the risks are given as intervals, the width of the boxes representing risks to show these intervals, i.e. the left hand side of the box indicates the minimum likelihood of the interval, and the right hand side indicates the maximum likelihood.



R1: Attacker gains unauthorized access to core data base with customer data
R2: Attacker causes service unavailability
R3: Attacker gains unauthorized access to user data

Figure 12– Risk matrix

2.3.2 Test Scenario Prioritization and Selection

The main purpose of our approach is to identify and prioritize tests based on risk models. The outcome of the testing can be used as a means of increasing the confidence in the likelihood values of a risk model.

Given that risk values have been obtained from all risks in a risk model, we must first decide whether or not it is necessary to perform the testing, i.e. whether an increased confidence in the correctness of the likelihood values will have an impact on the decisions that are made based on the risk model. This can be determined by looking at the risk matrix visualization. To see what we mean by this, consider the risk matrix shown in Figure 13. The risk matrix is separated into two risk values (acceptable and unacceptable) by the diagonal line. Assume that risks classified as acceptable can be accepted without treatment, and that the unacceptable risks must be treated. We consider each of the three risks in turn, and ask whether or not the decision regarding its treatment is affected by acquiring new knowledge about the correctness of its likelihood value.

Risk A is fully contained in the acceptable area. Thus, assuming that the risk model is correct in the sense that actual real likelihood of the risk is within the likelihood interval of risk A then there is no need to obtain new information through testing to reduce the size of the likelihood interval. Similarly, risk B is fully contained in the unacceptable area and it can therefore be treated. Thus the decisions regarding treatment for risk A and risk B is not affected by our confidence in their likelihood estimates, and if these were the only risks of the risk matrix, then there would have been no need to perform any testing on the basis of the risk assessment. However, risk C is a different matter, it spans both the acceptable and the unacceptable area, thus the problem is not necessarily that the risk is unacceptable, but that we do not know whether it is acceptable or not. Regarding risk C, it makes sense to perform testing to gain new knowledge that allows us to determine whether C should be treated or not.

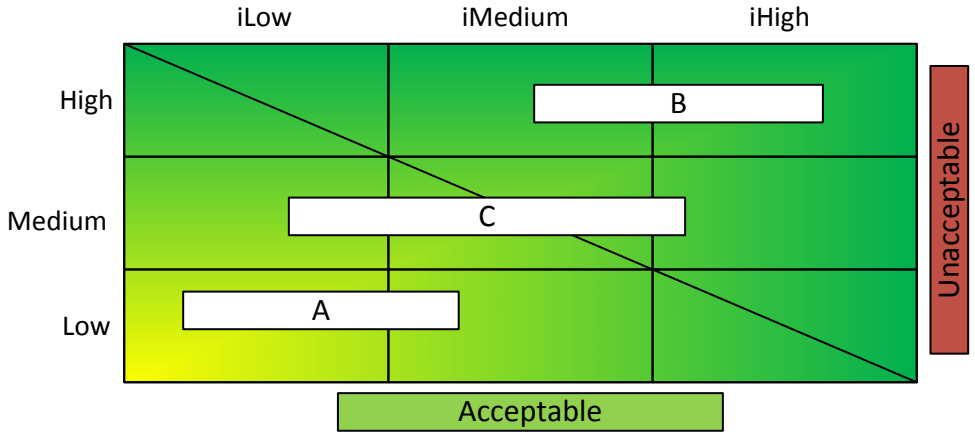


Figure 13 – Risk matrix with risk values

We have previously described a technique for test prioritization and selection based on a risk model. For a technical description of the technique, the reader is referred to [1] and [2]. For the purpose of self-containment we will only briefly describe the technique here and illustrate it with an example.

A risk model can be seen as a set of statements about the world. Testing a risk model corresponds to checking the degree to which these statements are correct. Given a risk model, the first question we must ask is which of its kinds of statements are the most natural starting point for test identification? As discussed further in [2], we believe that the statements derived from edges of a risk model are the most natural starting point. An edges going from a node A to a node B with conditional likelihood l means that "*A leads to B with conditional likelihood l*", thus a test procedure corresponding to this edge is a statement of the form "*Check that A leads to B with conditional likelihood l*".

Potentially, every edge of risk model gives rise to a test procedure. However, in practice we do not have time to test every one of these test procedures. Thus we have to prioritize and then select the test procedures that are most important. To achieve this, we must for each edge of a risk model ask whether its corresponding test procedure is within the scope of the risk assessment, and if yes, estimate the resources/effort it would require to implement and execute the test procedure. Subsequently, given an estimate of maximum total effort available for testing, we can use the technique described in [1] and [2] to obtain a prioritized list of test procedures that should be implemented and tested. In the following, we illustrate this in an example.

Assume we want to identify test procedures on the basis of the risk model shown in Figure 11. Our first task is to check whether the test procedure corresponding to each edge of the risk model is within the scope of the assessment, and if yes, estimate the time it will take to test it. Assuming that we are only interested in performing software security testing, we cannot check how often attacks are initiated in the first place. Thus the two outgoing edges from the Attacker are out of scope. Assume also that all edges after the threat scenarios describing successful attacks are regarded as out of scope. We are then only left with the two edges going from the threat scenarios describing attack initiation to successful attacks. Assume that we estimate the time it takes to implement and execute these tests procedures to 1 day each, and that we only have 1 day in total available for doing the testing. We are thus forced to choose which one of the test procedures to test. In order to decide this, we use our technique for test procedure prioritization described in[1] and [2], and we obtain the priority values shown in Table 8. Since test procedure id1 has a higher priority than test procedure id2, we choose id1 as the test procedure to test.

Test procedure	Priority
Id1: Check that Cross Site Request Forgery (aka Session Riding) leads to Cross Site Request Forgery (aka Session Riding) successful with conditional likelihood [0.001, 0.1], due to vulnerabilities OWASP Top Ten 2007 Category A5 - Cross Site Request Forgery (CSRF), Incorrect Permission Assignment for Critical Resource, Cross-Site Request Forgery (CSRF) and Improper Control of a Resource Through	2.138E-4

Test procedure	Priority
its Lifetime.	
Id2: Check that HTTP Response Splitting leads to HTTP Response Splitting successful with conditional likelihood [1.0E-4, 0.001], due to vulnerabilities Insufficient Comparison, Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting'), Improper Enforcement of Message or Data Structure and OWASP Top Ten 2007 Category A2 - Injection Flaws.	3.152E-8

Table 8 – Prioritized test procedures

3 Security Test Patterns

3.1 Security Test Strategies

Security test strategies constitute an abstract mechanism to specify a goal that shall be achieved by a test case. Whereas such a strategy specifies the goal, it does not define *how* it can be achieved. In this way, a strategy is a specification that has to be implemented. A test case has mainly two aspects, therefore there are two kinds of security test strategies:

- Stimulation, i.e. submitting inputs to the system under test. The goal of such a stimulus or a set of stimuli is to trigger a potential vulnerability. Often fuzzing is used to generate stimuli for security testing.
- Observation, i.e. checking how the system under tests behaves when stimulated with certain inputs. The observation of actual behavior of the system under test is compared to the expected behavior in order to determine the test verdict. While for functional testing the SUT's interfaces are observed, this might not be sufficient for security testing to detect whether a vulnerability has been triggered.

A stimulation strategy is a *specification* that a SUT shall be stimulated in such a way that a certain vulnerability may be revealed. The stimulation is generated by an implementation of such a stimulation strategy. There may be several implementations of a stimulation strategy. These implementations may differ from one to another, e.g. in the programming language or in the tools that are employed to provide appropriate test data or test cases. Several implementations in different programming languages allow for an easy integration in different testing tools that would take advantage of the security test pattern catalog by integrating the corresponding security test strategies' implementation of a pattern.

In addition to the name of a security test strategy, an interface description is provided. Each implementation of a certain security test strategy has to follow this interface. Such an interface description serves two purposes: On the one hand, it gives a tool provider the possibility to implement a strategy or to adapt the interface of an existing test generator to a strategy's interface. On the other hand, such an interface specification enables a tool developer the integration of test strategies' implementations by using this interface. Since several implementations of a strategy in different (programming) languages are possible, it seems not useful to provide the interface description itself in one programming language. Furthermore, the implementation of a test strategy can be in natural language to support manual usage of a test pattern. Hence, a programming language seems not appropriate for the interface description. Therefore, the interface description is provided in natural language, where each parameter and the return value are declared and accompanied by an informal description in natural language.

The question how strategies are finally used is open. They can be directly used by a risk-based security testing tool that selects for each vulnerability, the corresponding security test pattern and starts test case generation by executing an appropriate implementation of a test strategy. This approach can be used to directly stimulate the SUT's interfaces by submitting the generated test data or whole test cases. The approach also allows for model-based testing. In this case, the corresponding test strategies have to be applied to a model of the SUT's behavior. However, the integration of a stimulation strategy in a tool has to be done manually with the help of interface description provided in natural language, while the actual stimulus generation is provided by an implementation of the strategy within a security test pattern.

3.1.1 Stimulation Strategies

Stimulation strategies describe what shall be achieved by stimulating a system under test. In the context of the RASEN project, this is the weakness that is referenced by and the target of a security test pattern and shall be triggered by stimuli. Therefore, an individual stimulation strategy is specified for each weakness.

There is a generalization hierarchy of security test patterns, e.g. there is a pattern *Improper Input Validation*[1] that is a generalization of the patterns for *SQL Injection*[1]and *Uncontrolled Format*

String[1]. The same hierarchy is constituted by the stimulation strategies of those test patterns that reference these strategies and is depicted by Figure 14. Therefore, a stimulation strategy of a general pattern subsumes the stimulation strategies of its specializing patterns.

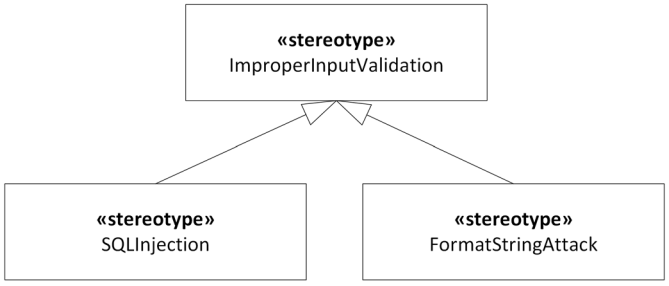


Figure 14 – Generalization hierarchy of stimulation strategies for data fuzzing (excerpt)

Depending on the goal, a stimulation strategy requires that either test data or whole test cases is generated. For instance, in order to detect a SQL injection vulnerability, corresponding input data that is able to manipulate a SQL query string shall be generated whereas for a replay attack vulnerability, a message sequence has to be generated (consisting of an authentication message, a logout message and the first authentication message again). Hence, appropriate input may be required by a strategy in order to generate the correct test data or test sequences. In case of the SQL injection vulnerability, this might be information about the database management system and the database schema, in case of the replay attack vulnerability, this is the authentication and the logout message. This is realized by the aforementioned interface description.

Test coverage items allow the automatic assignment of appropriate values to the strategies' interface once the items have been determined. They are described in Section 3.2.

In order to generate actual test cases or test data, a stimulation strategy is interpreted by a corresponding tool. In order to allow model-based test generation, we developed a UML profile that contains all the stimulation strategies in a generalization hierarchy, as described above using the example of SQL injection. Figure 15 shows an excerpt of this profile for data fuzzing stimulation strategies. These stereotypes refer to messages in UML sequence diagrams, parameters of the corresponding operation and, in case of data fuzzing generators, to value specifications, the arguments of a message. The reference means that this message argument shall carry fuzz test data generated by the applied stimulation strategy. The individual stimulation strategies' stereotypes has properties that correspond to the strategy's interface described above.

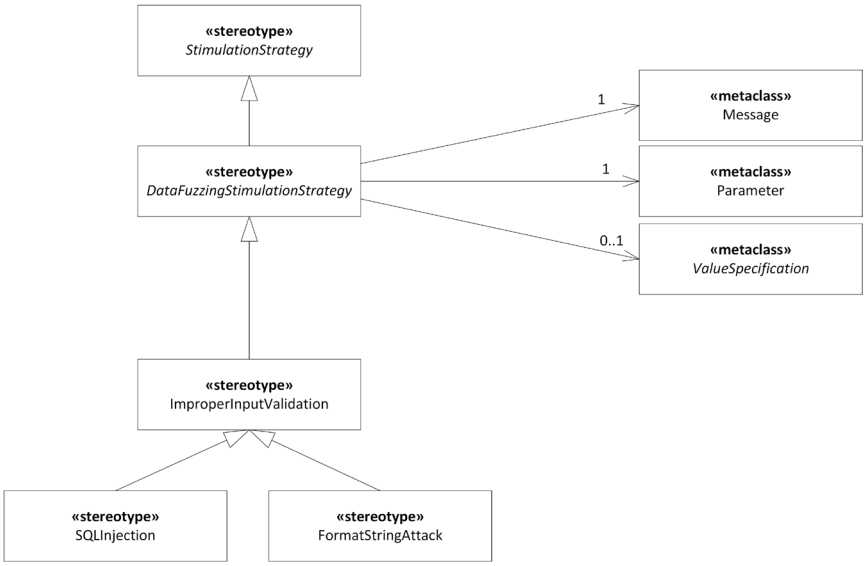


Figure 15 – Excerpt of UML profile for stimulation strategies for data fuzzing

Actually, we developed two separate UML profiles for stimulation strategies, one for the test design technique data fuzzing (Figure 15 shows an excerpt of it) and a second one for behavioral fuzzing. Section 5 described stimulation strategies specific

3.1.2 Observation Strategies

In contrast to stimulation strategies, observation strategies are used to determine the test verdict. In contrast to functional testing, the test verdict for security test cases cannot necessarily be determined by evaluating the immediate answer to a stimulus. A security-relevant weakness that is triggered by a stimulus does not necessarily show a characteristic behavior as a response to the stimulus. For instance, a successful SQL injection may add records to the database, e.g. a new user with administrative privileges. This new database record may be generated due to an SQL injection attack using a completely different functionality. Hence, this new record as a result of the SQL injection attack can only be observed if someone authenticates with this new user or an administrator calls the user management functionality. In case of a buffer overflow that injects code to the SUT, this code may be executed later. Stored cross-site scripting may be visible on a completely different page of a website.

Observation strategies take this into account by defining what modification or behavior of a system shall be looked for in order to determine whether a test case was successful in triggering a weakness. However, the kind of observation to be made on a system after possibly triggering a vulnerability may depend on the individual test case and may differ from test case to test case.

Considering SQL injection attacks, the result of such an attack may be an error message on the frontend of a web application, a bypassed authentication or a new database record. This depends on the actual SQL query part that is injected to the SUT and the database management system employed by the SUT (different database management systems have a slightly different SQL syntax, therefore the very same SQL injection string may result in a new record for one database system and a SQL syntax error for another system). Hence, each observation strategy suitable for a certain weakness (and thus, security test pattern) has to be applied to each test case and its implementation executed after the stimulation part of a test case in order to determine the test verdict. If one observation strategy detects the modification in question, previous stimuli to the SUT triggered a vulnerability and the test verdict is FAIL.

The simplest observation strategy, often used for fuzzing, is a connectivity check [8] that simply checks at the end of a security test case whether the SUT crashed by trying to connect to it. If attempt to connect fails, the SUT crashed or hanged and the test verdict is FAIL. Therefore, this observation strategy can only detect weaknesses with respect to availability of the CIA triad (confidentiality, integrity, availability).

Slightly more advanced is the so called valid case instrumentation that executes after each security test case a valid functional test case in order to determine whether the SUT behaves as expected and specified [7]. This strategy may lead to false positives as well as to false negatives. False positives occur if the functional test case employed for test verdict arbitration depends on the state of the SUT that may be changed from the previous stimuli without triggering a vulnerability. If the chosen functional test case tests a different part of the SUT than the one tested by the previous security test case, this may result in false negatives because a triggered weakness does not have an effect on the functionality involved in the functional test case. Therefore, valid case instrumentation has several pitfalls that make it less useful for security testing than required. It would lead to false negatives, i.e. actual vulnerabilities are not detected, and to false positives that require manual analysis although no vulnerability was triggered.

Less general observation strategies that are tailored to the weakness in questions seem reasonable and necessary. While different monitoring techniques are available, they are not related to vulnerabilities by themselves. This is where security test patterns can contribute to security testing by combining all the required information for testing for a certain vulnerability: the vulnerability itself, appropriate stimulation strategies that are able to trigger such a vulnerability and observation strategies that are tailored to the individual vulnerability to be detected, that are more specific than general monitoring techniques such as simple connectivity checks or valid case instrumentation that might miss triggered vulnerabilities. Additionally, such vulnerability-dependent observation strategies

may reduce the effort for security testing. Only the mechanisms that are appropriate for certain vulnerabilities, i.e. mentioned in the corresponding security test pattern, are necessary to implement. Additionally, several observation strategies may be appropriate across different vulnerabilities/patterns, and thus, reusing these strategies may also reduce the effort for effective test verdict arbitration. For instance, a cross-site scripting attempt as well as an SQL injection attempt may lead to error messages in the SUT's frontend (user interface) or backend (log files). Therefore, tests for several vulnerabilities may benefit from an implementation of the corresponding observation strategy without losing the precision of the verdict arbitration. Additionally, they may also allow a precise verdict arbitration that can be performed automatically.

However, an implementation may strongly depend on the SUT's implementation. Checking for added records in a database depends on the installed database management system, its credentials required to access the database and the machine that the database hosts. While completely universal implementations of observation strategies are sometimes prevented, the abovementioned parameters for the security test strategies may overcome this impediment partially. These parameters can be used to give an observation strategy the necessary information to access the database and check its tables for newly added records. Thus, an implementation catalogue of observation strategies can be established that provides either generic implementations with parameters whose appropriate values are assigned during test pattern instantiation (see Section4) while some observation strategies may be generic without any parameters, e.g. checks for an error message in the frontend of a web application. However, there will still remain cases where an individual implementation of observation strategies is necessary. At least, when the implementation of an observation strategy for a certain system is done, the approach still enables automated security test generation and execution.

3.2 Test Coverage Items

According to ISO29119-1, a test coverage item is derived from a testable aspect of a component or system by a test design technique and enables the measurement of test execution [9]. Generally, test coverage items can be divided into two categories: they can be a part of the SUT, i.e. an interface, a protocol, a certain message or message parameter, or they can result from a test design technique and thus, constitute the test case space. For instance, fuzz testing can be used to determine test coverage [9].

Test coverage items serve two purposes: they are used as input for test purposes and test strategies for test case generation and as input for metrics. Items that are part of the SUT may serve as input for test purposes that are used for test sequence generation as basis for data and behavioral fuzzing (Section4.2) and may be employed by test strategies for data and behavioral fuzzing (Sections 3.1 and 5). Items that are part of the SUT and those that result from the test design technique may serve as input for test metrics, in particular test coverage metrics, of course (Section6).

While the test coverage items that result from the test design technique can be automatically obtained by executing a security test strategy, this is not necessarily the case for those items that are part of the SUT. For instance, the interfaces that may be exercised in order to test for a weakness have to be determined manually. This can be done with tool support either in the risk model where vulnerabilities are mapped to SUT's interfaces or during test pattern instantiation.

3.3 Revised Security Test Pattern Description

The security test pattern template is updated according the previous refinements resulting in the changes described in Table 9

Original Field	Change	Rationale
Solution	removed manual solution	The manual solution is realized by an implementation of the stimulation and observation strategies.
Solution	added field observation strategies	As described in Section3.1.2, observation strategies are used to determine the test verdict and thus, are a natural supplement for a test pattern.

Original Field	Change	Rationale
Test Coverage Items	subdivided into several field	Test coverage items are, according to Section3.2, divided into SUT dependent and test design technique dependent items divided. This facilitates their determination by the user and their usage for security testing metrics.
Test Data	removed	Test data are generated by an implementation of a stimulation strategy, and thus, removed.
Testing Tools	removed	Test tools are referenced by an implementation of a stimulation strategy, and thus, removed.

Table 9 – Overview of Changes to 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	**Test Design Technique**	*Test design technique that is able to find the weakness.*
	Stimulation Strategies	*Stimulation strategies specify the goal to be achieved by generated test cases.*
	Observation Strategies	*Observation strategies specify what shall be checked after test case execution in order to determine the test verdict.*
	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).*
Test Coverage Items	**SUT dependent**	*Description of test coverage items in natural language that describe parts of the SUT that shall be exercised by test cases generated from this pattern.*
	Test Design Technique Dependent	*Description of test coverage items in natural language that describe those that are relevant for test metrics.*
Metrics	*Appropriate security testing metrics. See Section6.*	
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.*	
Specializations	*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 10 – Revised Security Test Pattern Template

3.4 Security Test Patterns

According to the new test pattern description, we updated the existing test patterns presented in [1] and added some new patterns addressing vulnerabilities from the OWASP Top 10 vulnerabilities [10].

3.4.1 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.[20]	
Solution	**Test Design Technique**	Data Fuzzing
	Stimulation Strategies	ImproperInputValidation (subsumes all the stimulation strategies of the specializations of this pattern)
	Observation Strategies	• Check for error message
	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.
Test Coverage Items	**SUT dependent**	• All interfaces of the system under test that get input from the external world, including the kind of data potentially exchanged through those interfaces [40] • User input fields
	Test Design Technique Dependent	• 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. [36	
Specializations	• SQL Injection • Uncontrolled Format String	
References	• OWASP Top 10 (2013): A1-Injection[35] • CWE-20: Improper Input Validation[20] • CAPEC-152: Injection (Injecting Control Plane content through the Data Plane)[18]	

Table 11 – Security test pattern "Improper Input Validation"

3.4.2 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.[21]	
Solution	**Test Design Technique**	Data fuzzing
	Stimulation Strategies	SQL Injection
	Observation Strategies	• Check database for new records • Check authentication state • Check for error message • Check for information disclosure
	Effort	Low to medium: can be highly automated using fuzzing techniques or SQL injection dictionaries.
	Effectiveness	Medium [21]to high, depending on detection capabilities by access to the affected database and to error messages
Test Coverage Items	**SUT dependent**	• Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a database query • User and hidden input fields • Names of tables and rows of the database schema • Values of existing records • Identifier of one record of each table
	Test Design Technique Dependent	• SQL injection payloads
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.	
Specializations	SQL Injection through a Database Abstraction Layer	
References	• OWASP Top 10 (2013): A1-Injection[35] • CWE-89: SQL Injection[21] • CAPEC-7: Blind SQL Injection[12] • CAPEC-66: SQL Injection[13] • OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005)[32] • OWASP: Automated Audit using SQLMap[39]	

Table 12 – Security Test Pattern “Improper Input Validation”

3.4.3 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.[28]	
Solution	**Test Design Technique**	Data Fuzzing
	Stimulation Strategies	SQLInjection
	Observation Strategies	• Check database for new records • Check authentication state • Check for error message • Check for information disclosure
	Effort	Medium to high
	Effectiveness	Medium
Test Coverage Items	**SUT dependent**	• Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a database query • User and hidden input fields • Names of tables and rows of the database schema • Values of existing records • Identifier of one record of each table
	Test Design Technique Dependent	• Database abstraction layer-specific injection payloads
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.	
Specializations		
References	• OWASP Top 10 (2013): A1-Injection[35] • OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005)[32] • CWE-564: SQL Injection: Hibernate[28] • OWASP Testing Guide: Testing for ORM Injection (OWASP-DV-007)[31]	

Table 13 – Security test pattern “SQL Injection through a Database Abstraction Layer”

3.4.4 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. [23]	
Solution	**Test Design Technique**	Data Fuzzing
	Stimulation Strategies	Format String
	Observation Strategies	• Check for messages containing memory dumps (differing substantially from regular messages received when valid messages were sent)
	Effort	Low: can be highly automated using fuzzing techniques and/or format string attack dictionaries.
	Effectiveness	Medium [23]to high, depending on detection capabilities by access to error logs and error messages
Test Coverage Items	**SUT dependent**	• 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
	Test Design Technique Dependent	• 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.[17]	
Specializations		
References	• OWASP Top 10 (2013): A1-Injection[35] • CWE-134: Uncontrolled Format String[23] • CAPEC-67: String Format Overflow in syslog()[14] • CAPEC-135: Format String Injection[17] • OWASP Testing Guide: Testing for Format String[33]	

Table 14 – Security test pattern “Uncontrolled Format String”

3.4.5 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.[26]	
Solution	**Test Design Technique**	Behavioral Fuzzing
	Stimulation Strategies	Missing Authentication (see Section5.2)
	Observation Strategies	• Check for successfully invoked function
	Effort	Low
	Effectiveness	High
Test Coverage Items	**SUT dependent**	• Interfaces that provide functions requiring authentication • Functions that require authentication • Authentication messages
	Test Design Technique dependent	• All sequences trying to access functions requiring authentication without prior 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.	
Specializations		
References	• OWASP Top 10 (2013): A7-Missing Function Level Access Control[37] • CWE-306: Missing Authentication for Critical Function[26]	

Table 15 – Security test pattern "Missing Authentication for Critical Function"

3.4.6 Authentication Bypass by Replay Attack

Pattern Name	Authentication Bypass by Replay Attack	
CWE-ID(s)	CWE-294	
Weakness Description	A capture-replay flaw exists when the design of the software makes it possible for a malicious user to sniff network traffic and bypass authentication by replaying it to the server in question to the same effect as the original message (or with minor changes). [24]	
Solution	**Test Design Technique**	Behavioral Fuzzing
	Stimulation Strategies	Capture Replay Attack (see Section5.1)
	Observation Strategies	• Check authentication state
	Effort	Low
	Effectiveness	High
Test Coverage Items	**SUT dependent**	• Interfaces that provide functions requiring authentication • Authentication messages
	Test Design Technique dependent	• All sequences trying to perform a capture-replay attack
Discussion	Capture-replay attacks are common and can be difficult to defeat without cryptography. They are a subset of network injection attacks that rely on observing previously-sent valid commands, then changing them slightly if necessary and resending the same commands to the server. [24]	
Specializations		
References	• OWASP Top 10 (2013): A2-Broken Authentication and Session Management[36] • CWE-294: Authentication Bypass by Capture-replay[24]	

Table 16 – Security test pattern "Authentication Bypass by Replay Attack"

3.4.7 Cross-Site Request Forgery

Pattern Name	Cross-Site Request Forgery (CSRF)	
CWE-ID(s)	CWE-352	
Weakness Description	The web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request. [27]	
Solution	**Test Design Technique**	Behavioral Fuzzing
	Stimulation Strategies	CSRF Attack (see Section5.3)
	Observation Strategies	• Check for successfully performed request
	Effort	Low
	Effectiveness	High
Test Coverage Items	**SUT dependent**	• Interfaces that provide functions requiring authentication • messages
	Test Design Technique dependent	• All sequences trying to perform a CSRF attack
Discussion	When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent, then it might be possible for an attacker to trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in exposure of data or unintended code execution. [27]	
Specializations		
References	• OWASP Top 10 (2013): A8-Cross-Site Request Forgery[38] • CWE-352: Cross-Site Request Forgery (CSRF) [27]	

Table 17 – Security test pattern "Cross-Site Request Forgery"

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. In the RASEN project, we would like to formalize the pattern part concerning the test intention, in order to produce automatically the corresponding test cases with model based testing.

3.5.1 Test Purpose Language

Within model-based testing, we propose to use a dedicated language, called Test Purpose Language. A test purpose is a textual expression, based on a regular expression that formalizes a test intention linked to a testing objective to drive the automated test generation on the behavioral model.
This language should also be close to natural language, in order to make it possible to understand the test objective easily, without prior test purpose language knowledge. The language description is detailed in the RASEN deliverable D4.2.1. [1]. The language enables to create several test objectives from a unique expression.

3.5.2 Test Purpose Example

In the RASEN context, we propose to use test purposes to formalize security test patterns. This allows the formalization of security test intention in terms of states to be reached and operations to be called.

As an example, the test pattern 2 [Appendix 8.1] concerning SQL Injection describes manual solutions based on attack pattern CAPEC-66. We can read in this Section:

Use the application, client or web browser to inject SQL constructs input through text fields or through HTTP GET parameters.

The idea is to inject SQL constructs on each input, and of each page of the application.

So we would like to obtain tests that navigate through the application to reach each of its pages, and then inject the SQL construct on each input field of this page. Each of those tests will be executed for each SQL construct provided by one of the Test Data tool specified in the Test Pattern, here Fuzzing Library Fuzzino [43].

This can be represented by a test purpose as shown in Figure 13.

```
for_eachinstance $param from
"Data.allInstances()->select(d:Data|not(d.action.oclIsUndefined()))" on_instance sut,

useany_operationany_number_of_timesto_reach
 "SUT.allInstances()->any(true).webAppStructure.ongoingAction.all_inputs-
 >exists(d:Data|d=self)" on_instance $param then

use threat.injectSQLi($param) then

useany_operationany_number_of_timesto_reach
 "self.webAppStructure.ongoingAction.oclIsUndefined()"
  on_instance sut then

use threat.checkBlindSQLi()
```

Figure 16 – Test Purpose sample for SQL Injection

3.5.3 Test Purpose Catalog

This test purpose is stored in a Test Purpose catalog (in xml format), with a reference to the pattern it belongs to. During the test generation process, regarding the information present in the CORAS diagram, the corresponding test pattern is chosen and the correct test purposes are selected for the test generation.

The test purposes must be generic and applicable to any SUT model, which are created using UML within the RASEN technology. For this reasons, some rules have to be respected when creating UML models of the SUT. For instance each input (text fields, http parameters, ...) must be represented as an instance of the “Param” class, each page as an instance of the “Page” class, and so on. To make it easier to create such a model, a domain-specific language (DSL) has been created.

3.5.4 DSL for Test Model Creation

The wizard presented in Section 4.1 enables to generate a UML model describing the SUT from a file in the DSL format.

This DSL is designed to capture the minimal information needed to navigate through all the SUT pages, and for each page the actions and inputs it contains, and the navigations between the pages, as shown in Figure 14.

```
PAGES {
        "PAGE1":INIT {
                ACTIONS {
                        "ACTION1" ("PARAM1" = "value1" => "PAGE2", "PARAM2" = "value2")
                                -> "PAGE2",
                        "ACTION2" ( …)
                                -> "PAGE3"}
                NAVIGATIONS {
                        "GOTO_PAGE4"
                                -> "PAGE4"}
        }
        "PAGE2" { …
        }…
}
```

Figure 17 – DSL structure for test model creation

An example from the Medipedia Use Case can be found in Appendix 8.2.

The UML model of the SUT generated from the presented DSL can be completed later to add more information and to specify further functional constraints about the SUT's expected behavior. However, it can be directly used as it is to generate test cases from the selected test purposes.

4 Instantiating Test Patterns for Test Case Generation

When test patterns relevant for the analyzed risks are selected and prioritized, test cases have to be generated. This is called test pattern instantiation. During instantiation, all the relevant values are determined and used to prepare and subsequently perform the test case generation. Currently, the instantiation process consists of four steps.

1. **Identification of test coverage items**

   Test coverage items that are part of the SUT are determined in this step. The items are taken from a system model of the SUT. Since the identification of test coverage items has to be performed manually (see Section 3.2), it should be supported by a tool. The natural language item descriptions also support the user in how to determine appropriate model elements. The identified items serve as input for step 2. Test coverage items are described in Section 3.2.

2. **Application of stimulation strategies**

   In this step, the identified test coverage items determined in step 1 are used to annotate them with the stimulation strategies referenced. For this purpose, we developed a UML profile that contains all the supported stimulation strategies. Its stereotypes are applied to the system model and reference the identified coverage items. Stimulation strategies are described in Section 3.1.1.

3. **Test sequence generation**

   In this step, test sequences as UML sequence diagrams are generated from the behavioral description of the system model. Generic test purpose definitions are supplemented with the elements annotated with stimulation strategies that shall be part of the test sequences. The test sequences constitute valid message sequences and test data. This process is described in Section 4.2.

4. **Behavioral fuzz test case generation**

   On the basis of the generated test sequences, behavioral fuzzing is applied when corresponding stimulation strategies are referenced by the test pattern. As a result, behavioral fuzz test cases are generated as UML sequence diagrams. Corresponding stimulation strategies and their realization employing behavioral fuzzing are described in Section 5.

4.1 Overview of the Test Generation Process

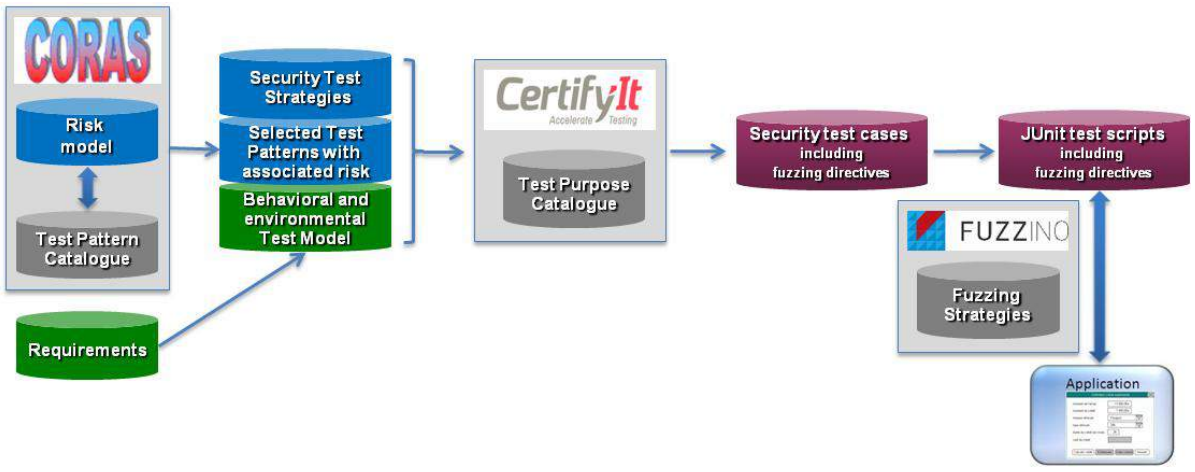


Figure 18– Test Generation Process

The process starts on the left at the risk model as a result from the risk assessment. As introduced in Section 2, a CORAS risk model (in relation with associated generic test pattern and vulnerability catalogues) enables to select security test purposes and to prioritize them regarding risk estimation. Security test patterns express the testing procedures of recurring problems in security testing and drive the risk-based security test generation. Figure 19 shows an example of such a CORAS risk model describing the SQL Injection threat scenario.

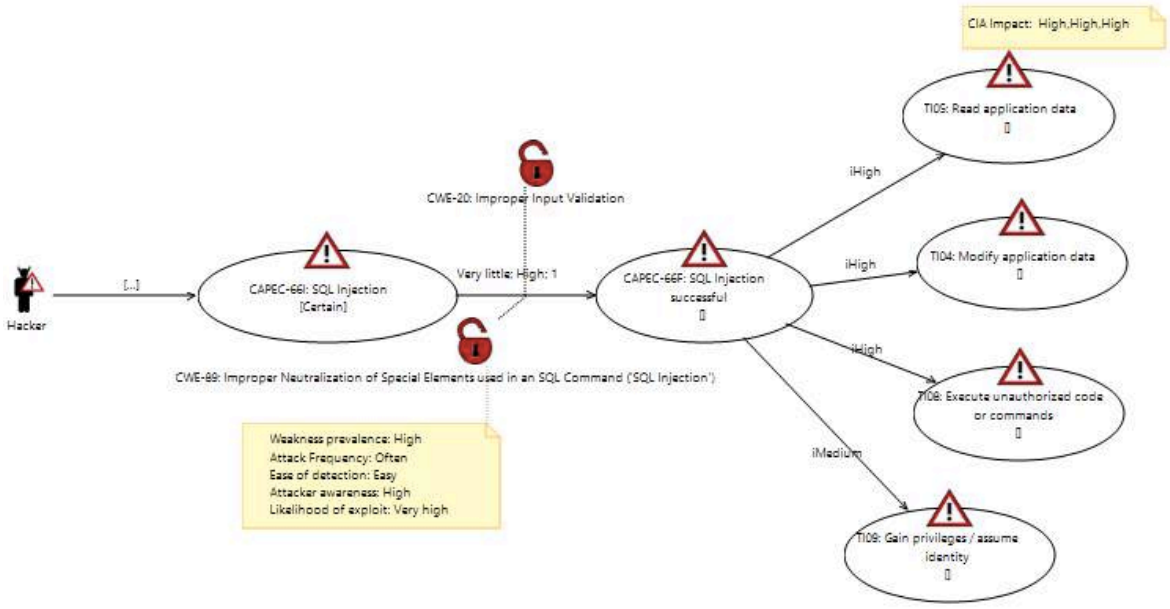
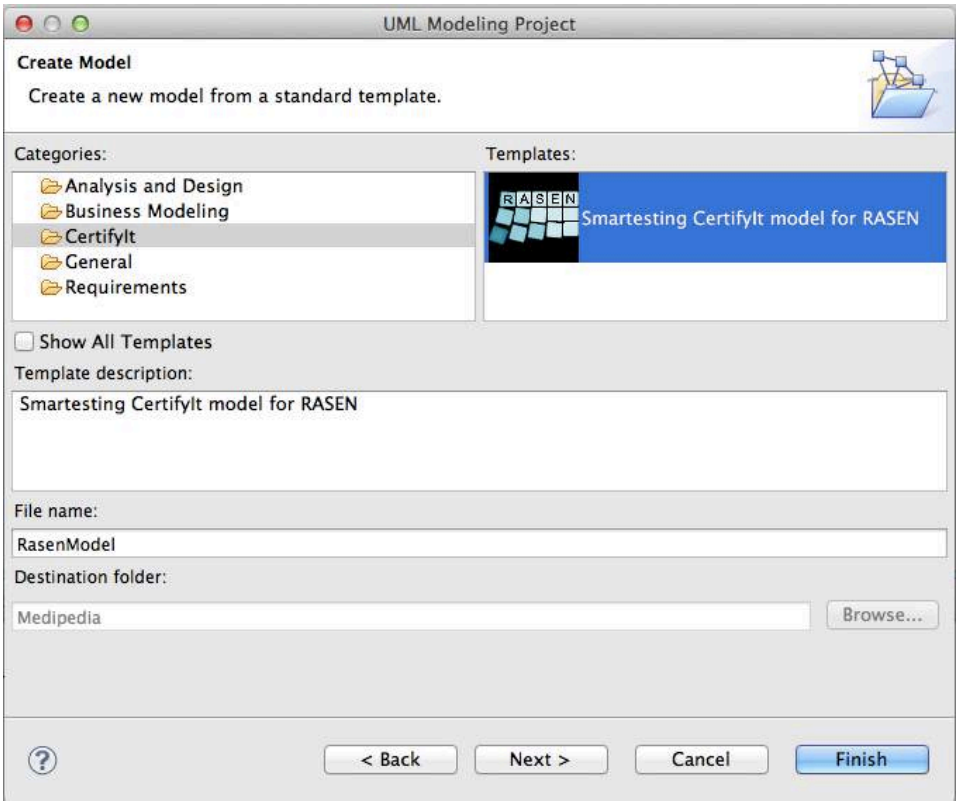
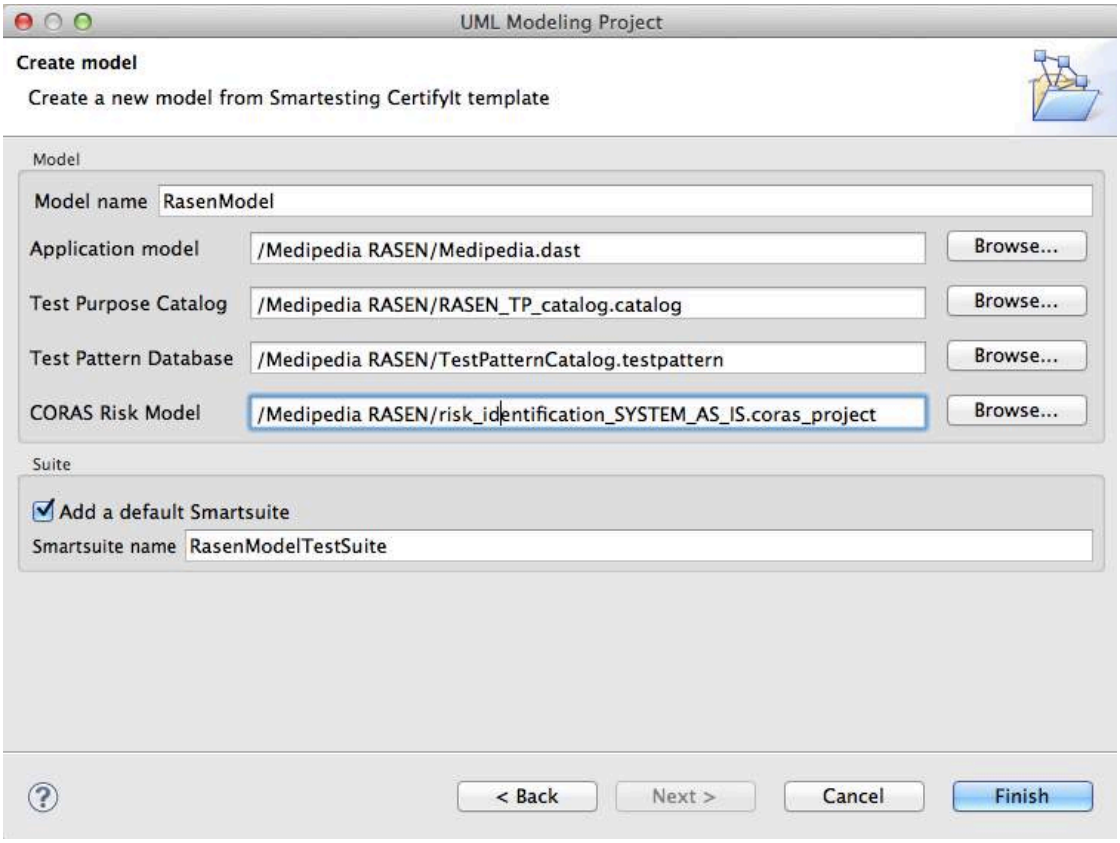


Figure 19 – Example of Risk Model from CORAS Tool



(a) Initialization of the Test Project



(b) Definition of the testing Artefacts to import

Figure 20– Configuration Screenshots to create the Test Project

Figure 20 depicts the screenshots of the RASEN tooling to initialize the test project. It allows creating all of the artefacts needed to apply the RASEN security testing strategies in a single environment:

- Each identified threat scenario of the CORAS risk model is linked to a dedicated security test pattern (introduced in Section 3) that defines the testing procedure allowing the detection of the corresponding threat in the system under test. This way, the security test patterns that have to be used by the test generation algorithm are gathered from the threat scenarios of each CORAS model related to the SUT. Moreover, likelihood and consequence are also collected from the CORAS model to assign a priority to the threat scenarios, and thus to prioritize them.
- To apply Model-Based approach, a UML test model is generated, using the DSL introduced in Section 3.5.4, to represent the Web application to be tested. Concretely, a UML test model consists of a class diagram to represent the static view of the system (with classes, associations, enumerations, class attributes and operations), a UML Object diagram to list the concrete objects used to compute test cases and to define the initial state of the system under test, and finally a state machine annotated with constraints (the Object Constraint Language specified by the OMG) to specify the dynamic view of the system.

For example, Figure 21 illustrates the resulting state machine of the SUT, which is automatically generated from the DSL instructions. This diagram graphically represents the behavioral aspect of the INFOWORLD "Medipedia" case-study, by modeling the navigation between pages of this Web application. States model Web pages, and transitions model the available links between these Web pages (HTML links, form submissions, etc.). Triggers of transitions are the UML operations of the class diagram. Guards of transitions (specified using OCL) precisely define the execution context of the transition. Finally, effects of the transitions (also specified using OCL) precisely describe its expected behavior that should be modeled for vulnerability test generation.

This model is also annotated by the test purpose identifiers and related information about prioritization and test procedure to be applied, which are inherited from the CORAS risk model and linked to test pattern catalog.

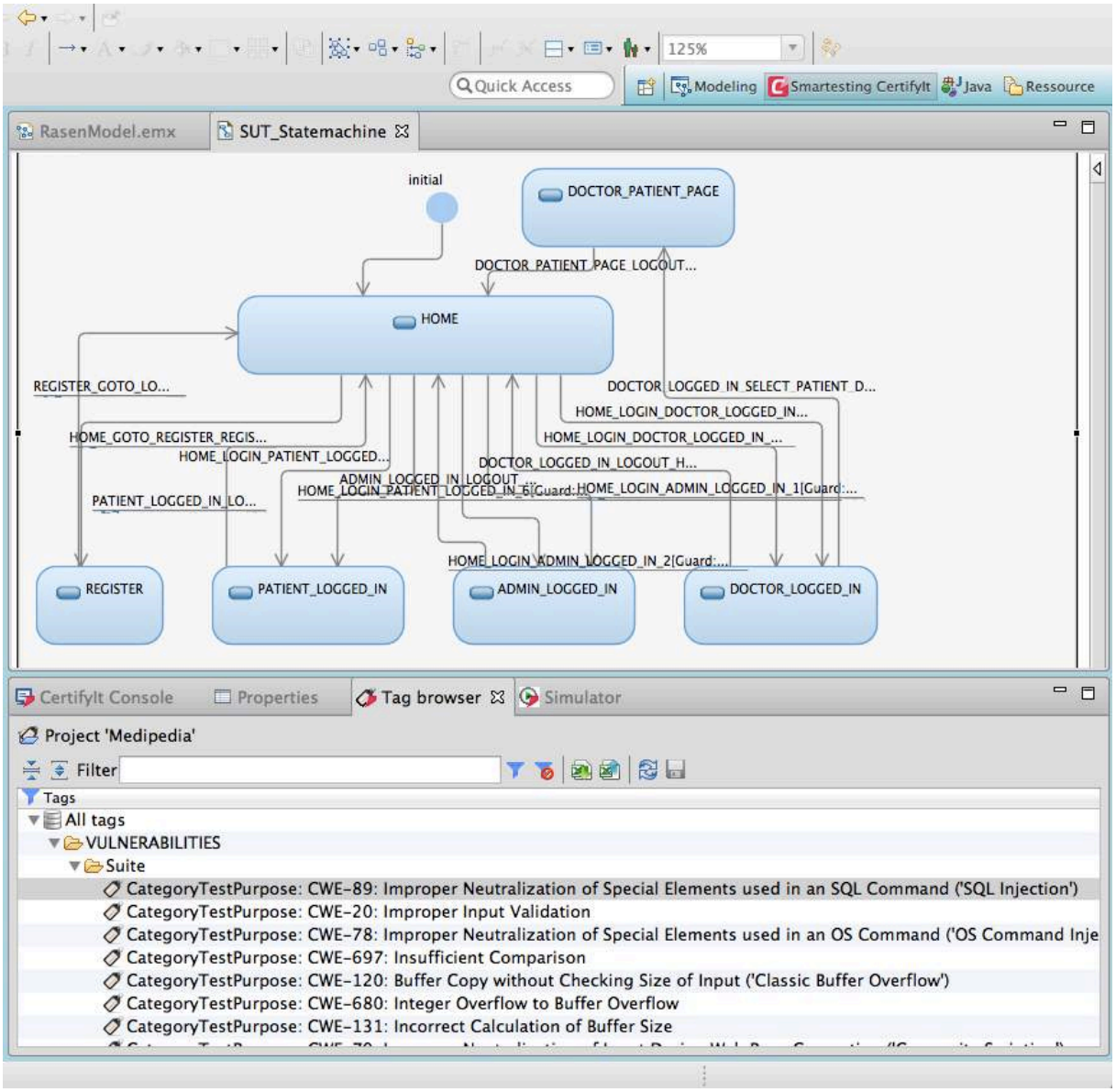


Figure 21 – State Machine of the SUT annotated by Test Purpose Identifiers

The next subsection introduces the test generation approach that is used to derive the security test cases from the test model and security test purposes.

4.2 Test Sequence Generation based on Security Test Purposes

Security test patterns based on prioritized vulnerabilities from the CORAS risk model thus provide the starting point for security test case derivation by providing information how appropriate security test cases can be created from risk analysis results. Therefore, as shown in Section 3 and depicted in Figure 22, dedicated and generic test purposes make it possible to formalize each targeted vulnerability imported from the test pattern catalogue. Hence, the test purpose, which is a high level or regular expression that formalizes a test objective (in terms of states to be reached, behaviors to be activated and operations to be called), is used to drive the automated test generation on the behavioral test model of the application under test.

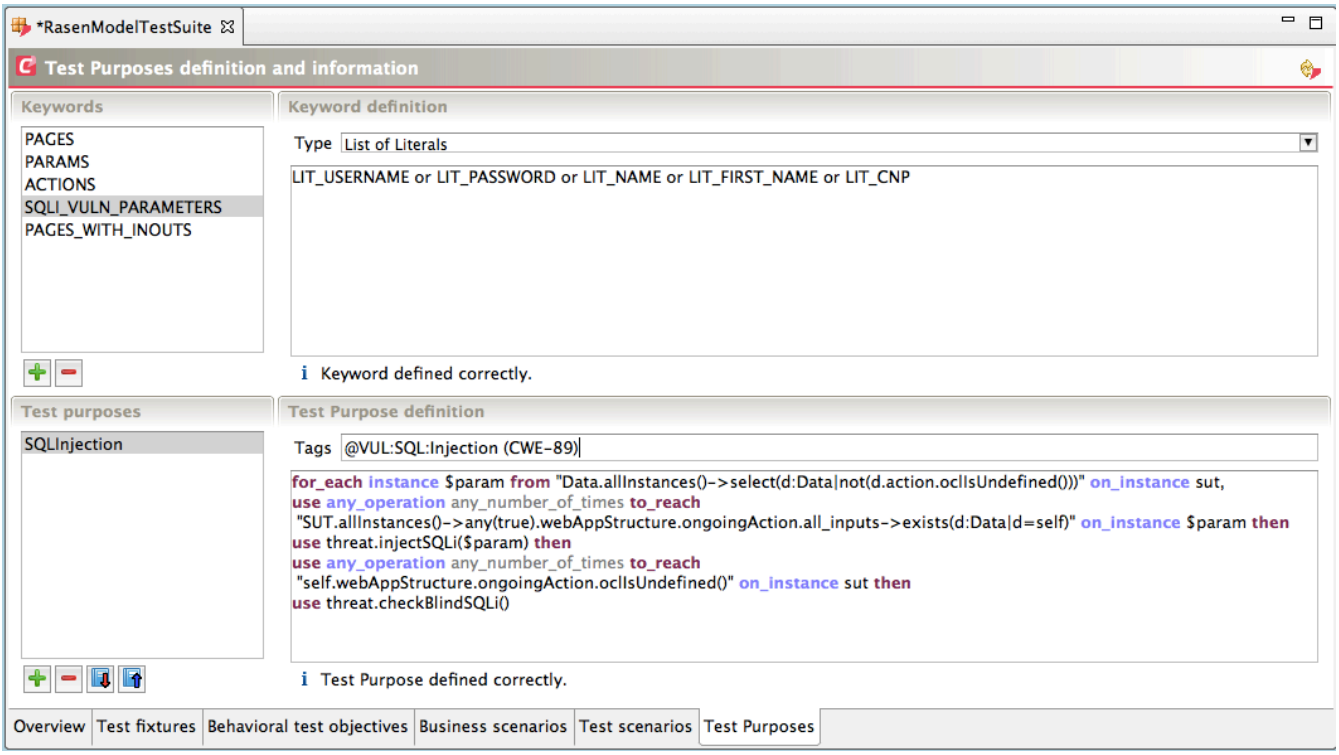


Figure 22 – Test Purpose Definition and Information

Each test purpose produces one or more abstract test cases verifying the test purpose specification and the behavioral test model constraints. As shown in Figure 23, such a test case takes the form of a sequence of steps, where a step corresponds to an operation call representing either an action or an observation of the system under test. It also embeds the security test strategies (from security test patterns) that is next used to apply data and behavioral fuzzing strategies during test scripts generation and execution.

Each test case can also be described and handled using UTP sequence diagrams.

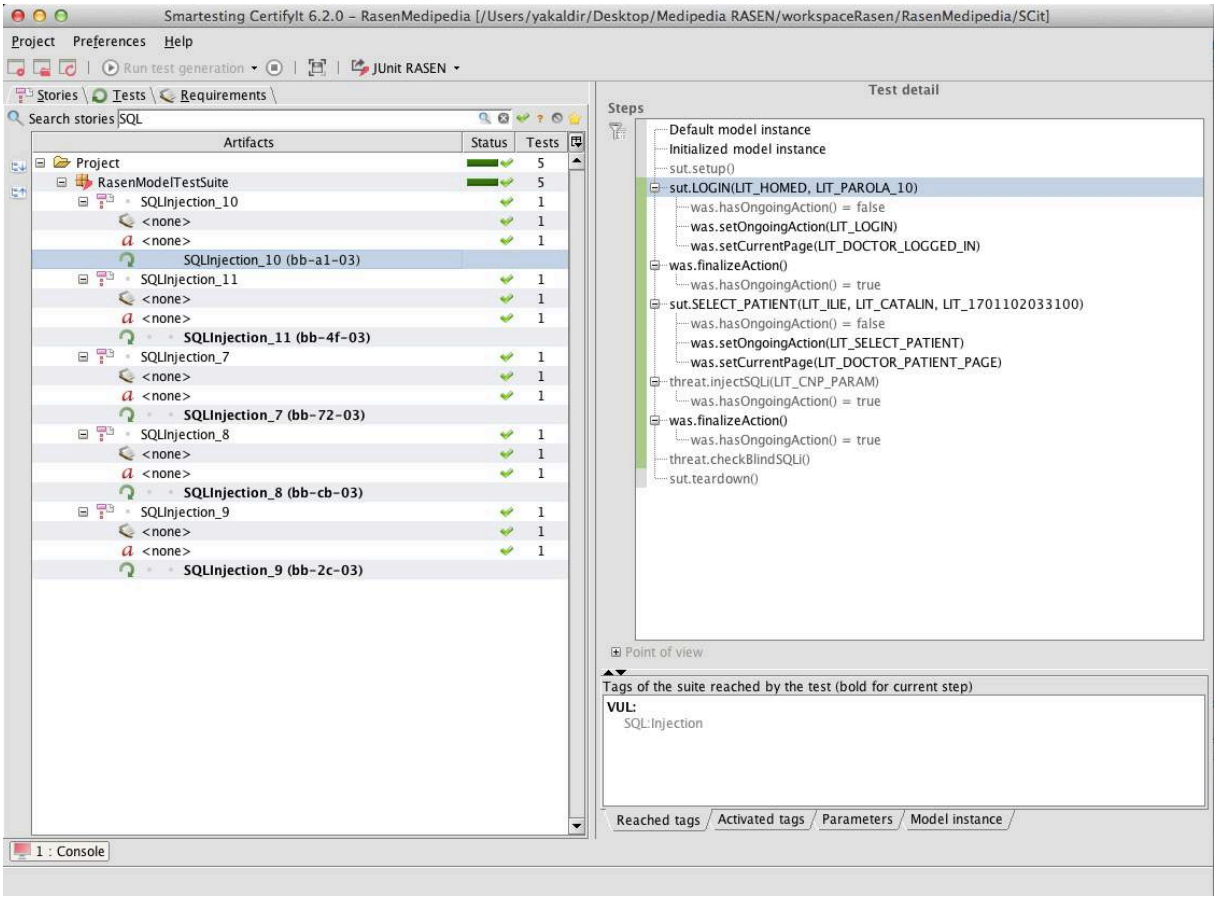


Figure 23 – Example of Abstract Test Cases Generated for SQLI

4.3 Test Case Execution

The last phase consists of exporting and executing the test cases in the execution environment. In our case, it consists of creating a JUnit test suite, where each abstract fuzzed test case is exported as a JUnit test case, and creating an interface. This interface defines the prototype of each operation of the application and links the abstract structures / data of the test cases to the concrete ones. Figure 24 shows a JUnit environment supporting the management of the executable test suite and its computation to assign the execution verdict.

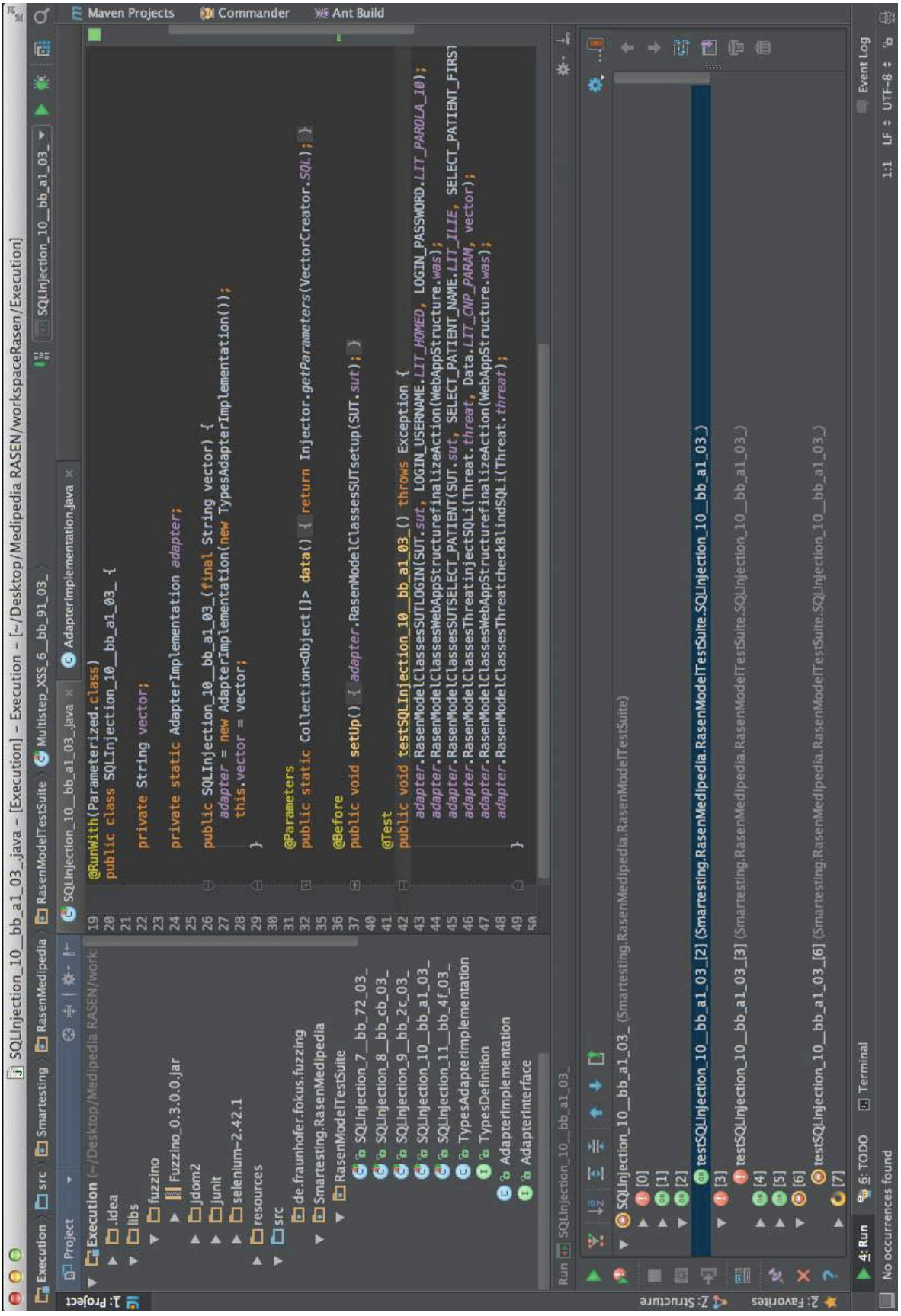


Figure 24– Screenshot of test execution using a JUnit environment

Since this process ensures the traceability between the verdict of the test case execution and the targeted vulnerabilities identified during risk assessment, the test results are gathered to automatically complement the risk picture of the system under test.

This overall testing process has been experimented using the industrial case-studies proposed by the RASEN project partners (Software AG, InfoWorld and Evry), and has proven to be successful regarding risk assessment guidance. However, these experiments have to be carried on to confirm the relevance and benefits of the approach to address large scale systems. For the last year of the RASEN project, in order to demonstrate such a scalability, we will evaluate the capacity of the approach to support a compositional testing strategy. On the one hand, this strategy is based on the composition of the test model and directives, which should enable to aggregate and re-use results from sub-models when addressing a global system. On the other hand, the use of the DSL, introduced in Section 3.5.4, should help and ease engineers to deal with large-scale systems by providing a simplified language to incrementally specify the SUT. These features, combined with the high level of automation of the RASEN techniques (about risk assessment guidance), should offer an iterative and incremental approach to perform risk-based testing for large-scale systems.

5 Behavioral Fuzzing for Security Testing

Behavioral fuzzing consists in mutating a valid message sequence to an invalid one by, for instance, removing or moving messages or modifying control structures (e.g. if-conditions, loop bounds). For this purpose, dedicated operators have been developed for UML sequence diagrams [D4.1.1, Behavioral fuzzing operators for UML sequence diagrams]. However, an invalid message sequence generated by applying a single behavioral fuzzing operator generally does not necessarily address a certain weakness. We explain how behavioral fuzzing operators can be composed such that a certain weakness is addressed on the example of the OWASP Top 10 weakness “broken authentication and session management”. Employing compositional behavioral fuzzing operators narrows the scope of behavioral fuzzing to certain weaknesses and thus, reduces the number of test cases to be generated. This goal can also be achieved if the elements to which behavioral fuzzing operators are applied are constrained.

Therefore, we considered OWASP Top 10 vulnerabilities, and selected those vulnerabilities that may be revealed by behavioral fuzzing. In the next step, we looked at the actual message sequences that triggers these vulnerabilities and how these message sequences can be constructed from a valid message sequence by behavioral fuzzing operators.

The result of this process are stimulation strategies for the OWASP Top 10 vulnerabilities and their realization using behavioral fuzzing operators for UML sequence diagrams where several operators are composed and the elements to which they are applied are constrained. Additionally, there are often several ways to address a certain weakness depending on the weakness and the way the protocol is modelled (see Section5.2).

5.1 Broken Authentication and Session Management (OWASP Top 10 A2)

This item of the OWASP Top 10 subsumes several weaknesses such as authentication bypass issues (CWE-592) and session fixation (CWE-384). An authentication bypass issue may be a capture-replay attack (CWE-294). This means that a valid login request is captured by an attacker and replayed later on to gain access to a system without proper credentials.

Behavioral fuzzing is able to trigger such weaknesses from a valid message sequence containing a login message and a logout message. Applying the behavioral fuzzing operator “Repeat Message” to the login message and applying the “Move Message” operator to the repeated login message in order to move it after the logout message. This composition of behavioral fuzzing operators imitates a capture-replay-attack.

However, such an attack is usually performed by an attacker from another machine. Behavioral fuzzing operators developed so far are not able to cope with this fact. When performing behavioral fuzzing, a valid UML sequence diagram is used and behavioral fuzzing operators are applied to elements of this sequence diagram (i.e. messages and combined fragments) in order to generate an invalid one. The lifelines of such a sequence diagram represent valid actors, e.g. a client, and the system under test. Security testing that considers malicious users, known as attackers, are not represented by such a valid sequence diagram. However, simply adding a lifeline for the attacker is not sufficient because this attacker has to perform its attack by sending malicious messages or message sequences to the SUT. These messages may result from behavioral fuzzing.

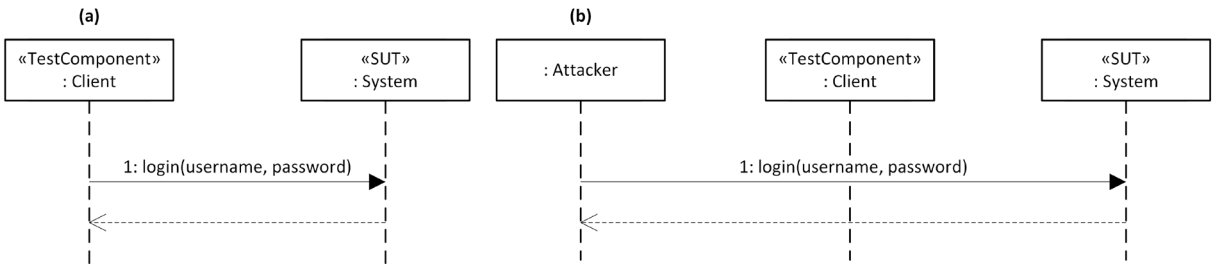


Figure 25 – (a) Valid message sequence and (b) resulting sequence after applying the operator “Move Message End to Other Lifeline”

A capture-replay attack is performed by e.g. sniffing network traffic of a valid user that interacts with the system the attacker would like to gain access. The sniffed network traffic is then replayed by the attacker using its own machine. This can be simulated by the attacker's lifeline in the sequence diagram and requires a new behavioral fuzzing operator that moves one end of a message to another lifeline. This is depicted in Figure 25: (a) shows a valid message sequence consisting of a single login message. When performing behavioral fuzzing for security testing, a lifeline representing the attacker is added (b). The behavioral fuzzing operator "Move Message End to Other Lifeline" was applied to the login message to move the sending message end to the lifeline that represents the attacker.

Using the new behavioral fuzzing operator, another composition of operators is possible in order to generate a capture-replay attack that brings the attacker into play: "Repeat Message" is applied to the login message and afterwards, the sending message end of the repeated login message is moved from the Client lifeline to the Attacker lifeline.

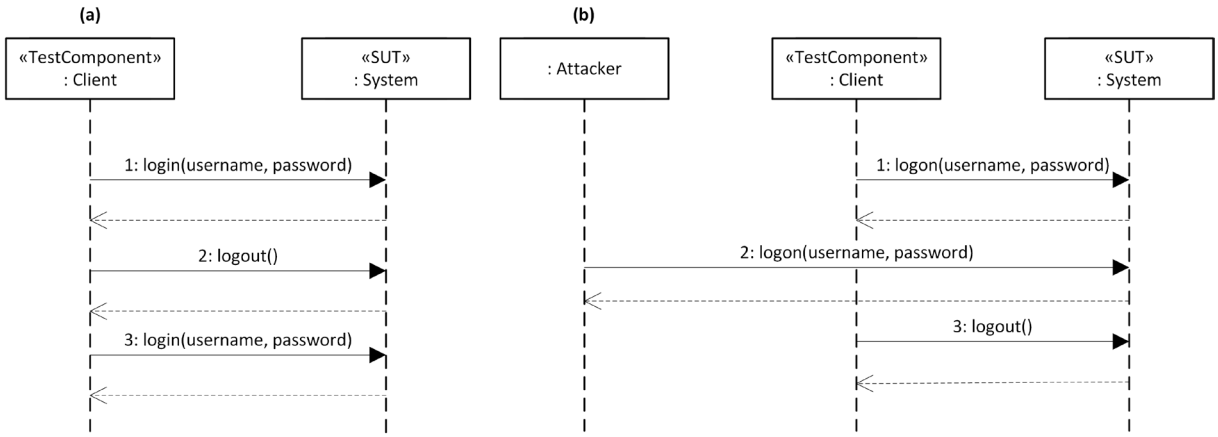


Figure 26: A capture-replay attack generated from applying (a) "Repeat Message" and "Move Message" and (b) "Repeat Message" and "Move Message End to Other Lifeline"

5.2 Missing Function Level Access Control (OWASP Top 10 A7)

This weakness addresses two aspects of access control: authentication and authorization. Weaknesses in the implementation may allow an unauthenticated user to get access to functions that require authentication or authenticated users to get access to functions for which the user has not the appropriate authorization. This may result if the authentication or authorization checks are implemented in a client and are not performed by the server. Therefore, such weaknesses may be revealed if the communication with server is not performed using the client application but directly on the protocol level. For instance, in case of a web application, an HTTP request to an authentication or authorization requiring function can be performed that cannot be done from the currently active web page. If the access control is not performed by the server, such an HTTP may be successful but must not.

Checking function level access control can be done using behavioral fuzzing by applying the operator "Remove Message" to the authentication message.

Authorization issues can be partially addressed by behavioral fuzzing, e.g. by inserting a message that requires higher authorization than the currently logged in user has, or by modifying combined fragments of a sequence diagram:

- The operator "Exchange Message" can be applied to the logon message and exchange it with another one with fewer privileges.
- If combined fragments are used to specify which functionally can be invoked depending on the authorization of a user, the corresponding guards can be modified by the "Negate Interaction Constraint" operator in case of a combined fragment of kind "optional", and by the "Exchange Interaction Constraints" in case of an "alternatives" combined fragment.

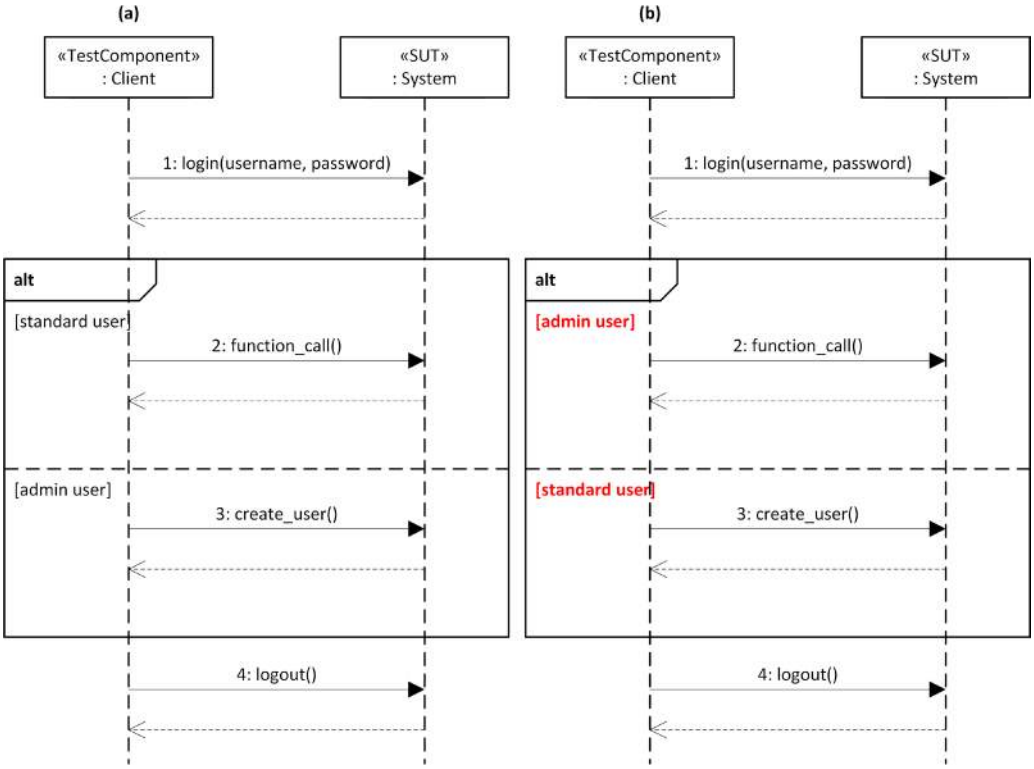


Figure 27: Checking for missing function level access control by applying "Exchange Interaction Constraints" to the alternative combined fragment (a) that tests whether a standard user can call administrator functionality (b)

5.3 Cross-Site Request Forgery (OWASP Top 10 A8)

Cross-site request forgery (CSRF) means that an attacker brings a user to make HTTP requests to a website that trusts a user. Thus, the attacker can perform requests with the credentials and the privileges of the user that performs as a proxy of the CSRF attack. This can be done by URL injected in an image-tag's source attribute of an HTML formatted e-mail that might be automatically followed by the e-mail client software. There have been several CSRF attacks in the past [41],[42]. A CSRF weakness can be triggered by the behavioral fuzzing operator "Insert Message" that uses e.g. an invalid token (see Figure 28) or no token, depending on the protocol.

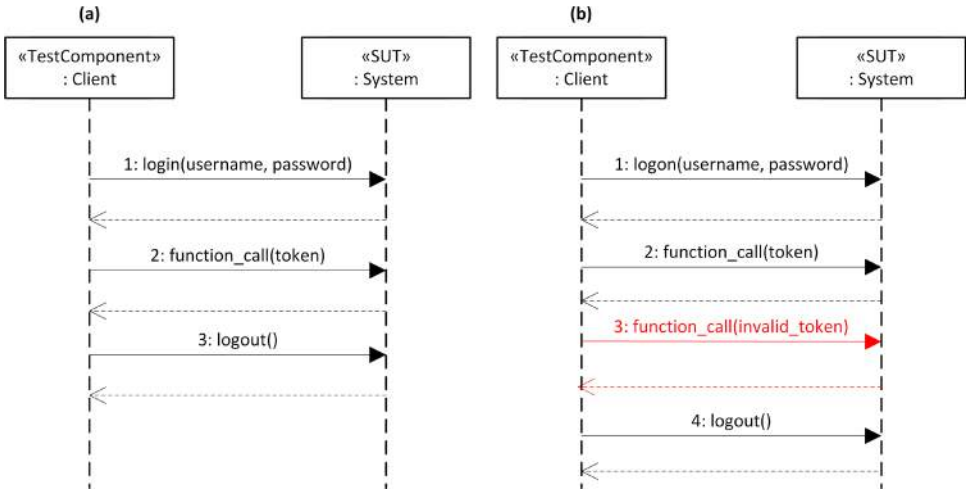


Figure 28 – A CSRF attack generated from a valid sequence (a) that calls a function by a valid token and (b) the resulting message sequence after "Insert Message" has been applied

6 Security Testing Metrics

Security metrics in general are described in [3]. Security testing metrics are a special case of security metrics providing generic functions that can be used to evaluate and interpret results from security testing.

Within the RASEN context, security testing metrics are a concept for the transfer of information from security testing to risk assessment. The results of the security testing metric functions should help to characterize the security risks of the system under test. Hence, it should be possible to improve a risk assessment based upon the results. Ideally, the functions of the security testing metrics yield risk assessment artifacts without requiring any further manual analysis effort or non-trivial conversions.

Taking test results and other information about the test and the system under test as input, these functions might for example be used to calculate likelihood values for the exploitability of some threat scenario. Testing metrics should support automation as far as possible. Unfortunately, there are no extensive catalogues with security testing metrics that are adequate for automation. Hence, we designed a new formal security testing metrics format and we started to build a new security testing metrics catalogue using that format.

6.1 RASEN Security Testing Metrics Format

Security testing metrics must have a name and a unique identifier. If some metric is altered in a way that it produces different results, then a new unique identifier has to be chosen for the altered metric.

Field		Description	Format
Identifier		Unique identifier, new for each version significantly changed	Number
Name		Meaningful identifier	Text
Category		Generalization of what the metric can be used for	Text
[Parameter] $^{>0}$	Type	The type of the parameter	XSD or text (notation format, {code})
	Description	Information about the parameter	Informal XHTML
Return value	Type	The return type	XSD or text (notation format, {code})
	Description	Information about the return value	Informal XHTML
[Function] $^{>0}$	Notation	The notation format, e.g. the name of the programming language	Text
	Code	Function signature and function body	{In function notation}
Description for all Functions		Information about what the functions do	Informal XHTML
[Feedback] $^{\geq 0}$	User	Who gives the feedback	RFC 822 Address
	Rating	Overall rating from poor (0) to excellent (10)	Number
	Application	How has the metric been used or analyzed?	Informal XHTML
	Results	Description of the results	Informal XHTML
	Comments	User experiences	Informal XHTML

Table 18 – The RASEN security testing metric structured format

In contrast to the identifier, the name is not required to be unique. The name should be descriptive for the entire testing metric.

Each security testing metric has at least one input parameter. Input parameters are specified with type and an informal description of what the parameter is expected to be semantically.

Any security testing metric returns one single value, for which a type has to be specified and a description has to be provided. Since the type of the return value might be a complex type, it is possible to return any number of objects.

The actual function of a security testing metric might be noted in some programming language. It is possible and encouraged to give multiple different notations of the same function within a single metric. However, all functions within one metric must produce exactly the same results. Because all functions noted within one metric are required to produce identical outputs, there should be only a single common human readable description for all notations within each security testing pattern.

Optionally, security testing metrics might contain user feedback. Testing metrics should be shared and reused, thereby the community might provide vital information. Users might want to share what they did, what the results were and what their overall impression is. Note that adding feedback information does never require a new unique identifier for the metric because the functions of the metric still produce the same results.

Table 18 shows the full structured format of RASEN security testing metrics.

6.2 Application of Security Testing Metrics within the Combined Risk Assessment and Security Testing Process

In the RASEN project, we develop a combined process of Test-Based Security Risk Assessment (TBRA – sometimes also TBSR is used as acronym) and Risk-Based Security Testing (RBST) as shown in Figure 29. The security testing metrics are vital for the TBRA step between test execution and security risk assessment.

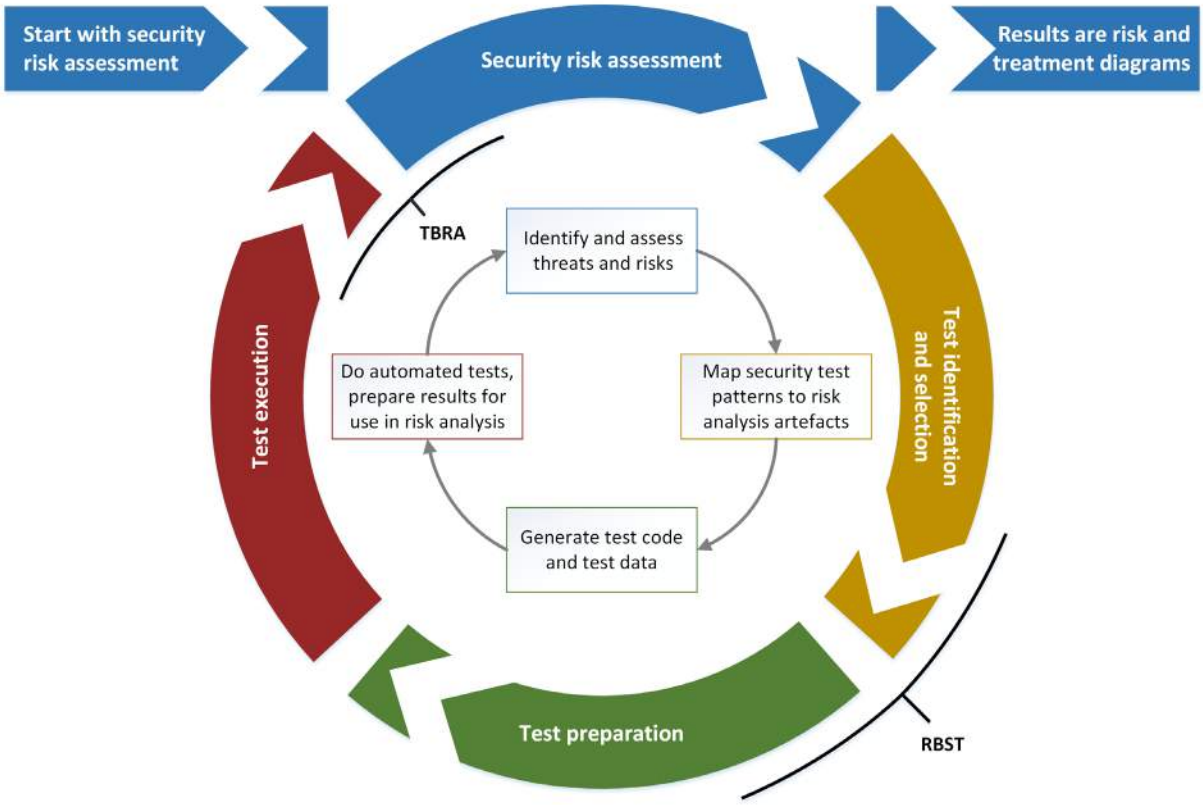


Figure 29 – Combined TBRA and RBST process

The RBST step between test identification and test preparation is basically done with the help of security test patterns and their relations to risk analysis artifacts as well as to system components. With the help of appropriate security test patterns, automated risk-based test case generation and test execution can be achieved. Test patterns also contain observation strategies, which can be used to collect raw test results.

Since the combined process includes Test-Based Security Risk Assessment, tests and the obtained test results should contribute to improve the risk assessment. Therefore, the tests and their results have to be further interpreted and evaluated. This is precisely where the security testing metrics come into play. Sound interpretation is highly dependent on the tests themselves. It depends on what was tested and how it was tested. That is basically the information that a test pattern contains. Therefore, the test patterns should indicate which security test metrics are most appropriate to analyze the testing process and the test results.

Each test pattern may suggest multiple applicable test metrics. For highly pattern specific metrics, the test pattern should contain the entire security testing metric. In contrast, for metrics that are more pattern independent, giving just the ID of the security testing metrics and keeping the metric separately so that it can be shared with other test patterns is preferable.

The risk analysts decide manually which metrics they want to use. Chosen metrics have to be instantiated, which should be possible at least semi automatically. Since all alternative function codes of each metric are required to produce equivalent results, it is possible to select the code format that fits best in the tool environment used for evaluating the function. Any function of a security testing metric will always require data for some input parameters. Typically, information used to instantiate and configure the test pattern has to be passed as input to the testing metric function. This includes especially, which elements of the risk model are actually tested. Additionally, any function of a security testing metric will expect information about the executed test cases and about the results that are observed with the help of the observation strategies of the test pattern. Some metric's functions might need further information, for example about the test execution time or about the entire time spend on testing including the instantiation of the test pattern. Security test patterns should contain information and even code snippets that help assigning the input parameters and calling the functions of the suggested metrics correctly. We refer to this as a bridge between a test pattern and a testing metric.

6.3 Categories of Metrics

6.3.1 List Up Metrics

These are the most basic kind of a testing metrics. Applying their functions does nothing but listing up a summary of the most important test results in a format specified by the metric. The results may be used as documentation in the risk graphs.

Additionally, list up metrics can be used to identify any unexpected incidents. These can be suggested as potential new unwanted incidents to the risk analysts.

6.3.2 Coverage Metrics

This kind of metric tries to calculate how complete the testing was. Such metrics measure for example, how much of the potential input value space has actually been created as test data or how much of the code of the system under test has in fact been executed during the testing process. Coverage metrics are widely used for all kinds of testing and there is a large amount of literature on that subject [6][5][4].

Coverage metrics are typically used as an indicator for the overall test quality. Results can be used for documentation purpose within the risk analysis. Eventually coverage of negative tests might be an indicator for the likelihood that some vulnerability exists at all.

6.3.3 Efficiency Metrics

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

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

6.3.4 Technical impact metrics

A technical impact metric tries to evaluate how much an entire system was affected by the testing process. Such metrics can be used to interpret the occurrence of multiple incidents in conjunction. Technical impact metrics are for example applicable to analyze the robustness against denial of service attacks.

6.4 Exemplary Security Testing Metric and its Instantiation

For demonstration, we present here an advanced high level efficiency metric. It analyzes the test results from an economical point of view.

Field		**Description**
Identifier		201
Name		Monetary_Feasibility
Category		Efficiency metric
Parameter Tests_Executed	Type	xsd:long
	Description	Total number of tests that were executed
Parameter Tests_Successfull	Type	xsd:long
	Description	Number of tests that did not trigger any unwanted incidents
Parameter Testing_Duration	Type	xsd:long
	Description	Duration of the entire testing process in milliseconds
Parameter Attackers_Funding_Base	Type	xsd:long
	Description	Optional. How much money would potential attackers spend?
Parameter Related_Risk_Artefact	Type	<xsd:complexType name="CBaseNodeInRiskGraph"> …
	Description	Optional. The extended CORAS risk analysis artefact the test pattern is associated with. Typically this is a threat scenario. Used only to calculate attackers funding base if parameter Attackers_Funding_Base is not specified.
Parameter Test_System_Costs	Type	xsd:long
	Description	Optional. The price of the technical systems used for generating and executing the test
Parameter	Type	xsd:long

Field		Description
Testing_Costs_Per_Hour	Description	Optional. Estimated running costs for testing one hour
Return value	Type	<xsd:complexType name="CLikelihoodFunction"> …
	Description	This is an expression of likelihood as a function over time. The function can be used to calculate for any given point of time the likelihood that the related unwanted incidents or faults will be triggered by attackers. The likelihood function expects a starting point of time as input and it returns some constant likelihood value as output.
Function	Notation	CSharp
	Code	…
Description for all Functions		First of all, the function calculates how long it takes to trigger an unwanted incident or fault. If no incidents have been triggered, then it is assumed that it will probably take twice the Testing_Duration. Let *T* be the calculated time it takes to trigger an incident with the system used for testing. Then the function calculates the price P_N indicating how expensive it is currently to trigger an unwanted incident, i.e.: $P_N = Test\_System\_Costs + Testing\_Costs\_Per\_Hour * T$ If the parameters are not specified, then default values Test_System_Costs = 1000 € and Testing_Costs_Per_Hour = 30 € are used. Due to technical progress, it is expected that the price will be halved at least every four years. Let *X* be the time between now and the beginning of the attack in years. Then the price P_X at the time now plus *X* years is: $P_X = P_N * 0.5^{X \div 4}$ Let *C* be the amount of money a potential attacker would probably spend. If the parameter Attackers_Funding_Base is set, then *C* is set to its value. Else if Related_Risk_Artefact is set, then the function tries to identify all the assets that might be affected if that risk artefact is exploited. The sum of the monetary values associated with these assets is in indicator for how much money probably attackers would be willing to spend. Hence, *C* is set to that sum. The function also tries to identify all human threats that might become attackers. If their financial capability is specified, then *C* is reduced to the highest financial capability of a potential attacker. If neither parameter Attackers_Funding_Base nor parameter Related_Risk_Artefact are set, then *C* is set to a default value of 3000 €. The returned likelihood function calculates time period *X* from its start time parameter and the point of time *N* for which P_N was calculated. It yields the following values:
Description for all Functions (continued)		**Low** if $P_X \div C > 1$

Field	Description	
	Medium	if $0.1 \leq P_X \div C \leq 1$
	High	if $P_X \div C < 0.1$

Table 19 – Efficiency metric for economic feasibility

The efficiency metric for economic feasibility shown in Table 19 can take benefit from the information that an extended CORAS risk graph can provide about the assets and about the human threats. However, since the one and only method specific parameter Related_Risk_Artefact is optional, the metric can also be used with other risk assessment methods.

Instantiation of the shown security testing metric can be completely automated. All non-optional parameters can be taken directly from the testing process. Values for Tests_Executed and Tests_Successfull can be obtained from the pattern code for test generation and test observation. Testing_Duration for completely automated testing is basically the test case generation time plus the test execution and observation time, so a value can also be produced with the help of the test pattern code itself. If CORAS is used for risk assessment, then the Related_Risk_Artefact parameter can be assigned automatically, too, since the test pattern is associated with some rick analysis artefact within the RBST process.

No doubt, the non-optional parameters can be taken directly from the testing process – but how exactly? For automation, there must be some kind of bridge, some mapping between the principally available information and the actual parameters of the security testing metric function which should be called. This bridge can be a code snippet that just passes values from the test pattern instance and observations from the actual test execution directly to the appropriate input parameters of the metric's function. Clearly, that bridge is test pattern specific and therefore it must be a part of the test pattern, rather than the testing metric.

The parameters Attackers_Funding_Base, Test_System_Costs and Testing_Costs_Per_Hour can only be set manually. These parameters are optional, but if the default values are not appropriate, then some manual action will be required for proper instantiation of the metric. Therefore, the test pattern specific bridge that should help automated metric instantiation must offer these optional parameters for optional manual input to the users.

Let **CTestPatternSpecificRawData** be a test pattern specific class that holds any raw data about the testing process collected during pattern instantiation, test case generation, test execution and test observation. The following C# code snippet could be used as a bridge for the efficiency metric economic feasibility function:

```
publicCLikelihoodFunctionCalculate_Monetary_Feasibility_Pattern_Specific_Bridge(
   CTestPatternSpecificRawData i_oTestPatternSpecificRawData,
   long ? i_nlAttackers_Funding_Base,
   long ? i_nlTest_System_Costs,
   long ? i_nlTesting_Costs_Per_Hour )
   {
      // call the pattern independent security testing metric function
      returnCalculate_Monetary_Feasibility(
         i_oTestPatternSpecificRawData.m_lTests_Executed,
         i_oTestPatternSpecificRawData.m_lTests_Successfull,
         i_oTestPatternSpecificRawData.m_lTesting_Duration,
         i_nlAttackers_Funding_Base,
         i_oTestPatternSpecificRawData.m_oBaseNodeInRiskGraphRelated_Risk_Artefact,
         i_nlTest_System_Costs,
         i_nlTesting_Costs_Per_Hour );
   }
```

6.5 Conclusion

With appropriate security testing metrics it is possible to do a more or less completely automated Risk-Based Security Testing and Test-Based Security Risk Assessment as described in D 3.2.2 chapter 4.1 (the RACOMAT method). However, the metrics need to be identified and instantiated. If the security test patterns contain bridges, then this can be solved without much manual effort.

The biggest problem at the moment is that there are only a few security testing metrics defined. The same is also true for the security test patterns – for many threat scenarios there are currently no existing appropriate security test patterns. Once there are extensive catalogues of patterns and metrics, then the concepts presented here will make the entire combined risk assessment and security testing process much easier.

Creating a library of good security test patterns and security testing metrics is not a trivial task. With the RACOMAT tool, we have now at least a tool that supports creating and editing testing metrics as well as test patterns. This allows us to create the metrics and patterns we need within the RASEN project. For the future, we plan to let an open community work with our pattern and metrics database, developing it further in collaboration. We expect that user feedback will be essential for quality assurance and for continuous improvement. Therefore, our testing metrics format integrates user feedback as a vital part of the metrics themselves.

7 Summary

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

In this deliverable, techniques for test procedure identification, prioritization and selection and test case derivation based on risk assessment results using CAPEC attack patterns were presented. It describes how a test procedure can be derived in three steps by (1) generating generic risk models from CAPEC attack patterns, (2) adapting them to the target of the risk assessment and (3) deriving from the target-specific risk model a test procedure consisting of select and prioritized test scenarios.

Based on these test scenarios, test patterns may be selected as starting point for test case derivation. They provide refined techniques for test generation (stimulation strategies) and test verdict arbitration (observation strategies). The actual test case generation starts by instantiating a security test pattern, employing security test purposes for test sequence generation and fuzzing techniques for actual security test case generation. In order to provide an overview of test progress and results, security testing metrics come into play serving as the first step from testing backwards to a refined risk model.

8 Appendix

8.1 Test Pattern sample: SQL Injection

```
<Test_Pattern ID="2"Name="SQL Injection">
<Effort>LowToMedium</Effort>
<Effectiveness>MediumToHigh</Effectiveness>
<!--optional attributes -->
<Test_Technique>Data fuzzing</Test_Technique>
<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.
        </Weakness_Description>
<Manual_Solution>Based on attack pattern CAPEC-66.&#xD;
    1. Use the application, client or web browser to inject SQL constructs input through text fields or through HTTP GET
       parameters.&#xD;
    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.&#xD;
    3. Check for error messages, delays, disclosed values in the client application and new/modified/deleted values in the
       database.</Manual_Solution>
<Effort_Description>Can be highly automated using fuzzing techniques or SQL injection dictionaries.
        </Effort_Description>
<Effectiveness_Description>Depending on detection capabilities by access to the affected database and to error messages.
      </Effectiveness_Description>
<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.</Discussion>
<Description_Of_Test_Coverage_Items>
<Test_Coverage_Item>Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a
            database query</Test_Coverage_Item>
<Test_Coverage_Item>User input fields</Test_Coverage_Item>
<Test_Coverage_Item>SQL injection payloads</Test_Coverage_Item>
<Test_Coverage_Item>Names of tables and rows of the database schema</Test_Coverage_Item>
<Test_Coverage_Item>Values of existing records</Test_Coverage_Item>
</Description_Of_Test_Coverage_Items>
<Generalization_Of>SQL Injection through a Database Abstraction Layer</Generalization_Of>
<Test_Data>SQL Injection Cheat Sheet</Test_Data>
<Test_Data>Fuzzing library Fuzzino</Test_Data>
<Test_Tool>Fuzzing framework Sulley</Test_Tool>
<Test_Tool>Sqlmap</Test_Tool>
<References>
<Reference ID="89"ReferenceLink="http://cwe.mitre.org/data/definitions/89.html"Title="SQL Injection"Type="CWE"/>
<Reference ID="7"ReferenceLink="http://capec.mitre.org/data/definitions/7.html"Title="Blind SQL Injection"Type="CAPEC"/>
<Reference ID="66"ReferenceLink="http://capec.mitre.org/data/definitions/66.html"Title="SQL Injection"Type="CAPEC"/>
<Reference ReferenceLink="https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OWASP-DV-
            005%29"Title="OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005)"Type="Other"/>
<Reference ReferenceLink="https://www.owasp.org/index.php/Top_10_2013-A1-Injection"Title="OWASP Top 10 (2013): A1-
            Injection"Type="Other"/>
<Reference ReferenceLink="https://www.owasp.org/index.php/Automated_Audit_using_SQLMap"Title="OWASP: Automated
            Audit using SQLMap"Type="Other"/>
</References>
<Test_Stimulation_Strategies>
<Strategy Name="SQL Injection"Description="A list of SQL injection test data will be generated by usage the fuzzing library">
<Parameter Direction="RETURN"Name="sqlInjectionData"Type="List of Fuzz Test Data"Description="A list of SQL injection test
              data"/>
<Process Language="Java">
<Data><![CDATA[
        public List<FuzzedValue<String>> generateSQLInjections() {

        StringSpecification stringSpecSQL = RequestFactory.INSTANCE.createStringSpecification();
        stringSpecSQL.setType(StringType.SQL);
        SQLInjectionsGenerator sqlInjections = StringGeneratorFactory.INSTANCE.createSqlInjections(stringSpecSQL,
                0);

        sqlIterator = sqlInjections.iterator();
```

```
                List<FuzzedValue<String>> sqlInjectionData = new LinkedList<>();
                int maxNumVal = 50;
                while (sqlIterator.hasNext() && maxNumVal-- > 0) {
                sqlInjectionData.add(sqlIterator.next());
                }

                return sqlInjectionData;
                }
]]></Data>
</Process>
</Strategy>
</Test_Stimulation_Strategies>
<Test_Observation_Strategies>
<Strategy Name="Check Database for new Records"Description="Login in database in SUT and check if new records are
                created with injection stimuli.">
<Parameter Direction="IN"Name="name"Type="string"Description="valid user name"/>
<Parameter Direction="IN"Name="password"Type="string"Description="valid password"/>
<Parameter Direction="IN"Name="sqlRequest"Type="string"Description="request for DB validation"/>
<Parameter Direction="RETURN"Name="newRecordsFound"Type="string"Description="'true' if new records are found, 'false'
                otherwise. If request could not be send, 'error' will be returned."/>
<Process Language="Java">
<Data><![CDATA[
                String newRecordsFound(String name, String password, String sqlRequest) {
                try {
                Connection con = connectToDB(name, password);
                Statement stmt = con.createStatement();
                ResultSet result = stmt.executeQuery(sqlRequest);
                return newEntryFound(result);
                }
                catch (Exception e) {
                return "error";
                }
                }
]]></Data>
</Process>
</Strategy>
<Strategy Name="Check Authentication State" />
<Strategy Name="Check for Error Message" />
<Strategy Name="Check for Information Disclosure" />
</Test_Observation_Strategies>
</Test_Pattern>
```

8.2 DSL file sample for the Medipedia Use Case

```
PAGES {
        "HOME":INIT {
                ACTIONS {
                        "LOGIN" ("USERNAME" = "admin" => "ADMIN_LOGGED_IN", "PASSWORD" = "parola-10")
                                -> "ADMIN_LOGGED_IN",
                        "LOGIN" ("USERNAME" = "admin2" => "ADMIN_LOGGED_IN", "PASSWORD" = "parola-10")
                                -> "ADMIN_LOGGED_IN",
                        "LOGIN" ("USERNAME" = "homed" => "DOCTOR_LOGGED_IN", "PASSWORD" = "parola-10")
                                -> "DOCTOR_LOGGED_IN",
                        "LOGIN" ("USERNAME" = "test_med2" => "DOCTOR_LOGGED_IN", "PASSWORD" = "parola-
10")
                                -> "DOCTOR_LOGGED_IN",
                        "LOGIN" ("USERNAME" = "hopac" => "PATIENT_LOGGED_IN", "PASSWORD" = "parola-10")
                                -> "PATIENT_LOGGED_IN",
                        "LOGIN" ("USERNAME" = "iliecatalin" => "PATIENT_LOGGED_IN", "PASSWORD" = "parola-10")
                                -> "PATIENT_LOGGED_IN"

                }
```

```
		NAVIGATIONS {
			"GOTO_REGISTER"
				-> "REGISTER"
		}
	}
	"ADMIN_LOGGED_IN" {
		NAVIGATIONS {
			"LOGOUT"
				-> "HOME"
		}
	}
	"DOCTOR_LOGGED_IN" {
		ACTIONS {
			"SELECT_PATIENT" ("NAME" = "ILIE" => "DOCTOR_PATIENT_PAGE", "FIRST_NAME" = "Catalin" => "DOCTOR_PATIENT_PAGE", "CNP" = "1701102033100" => "DOCTOR_PATIENT_PAGE")
				-> "DOCTOR_PATIENT_PAGE"
		}
		NAVIGATIONS {
			"LOGOUT"
				-> "HOME"
		}
	}
	"DOCTOR_PATIENT_PAGE" {
		NAVIGATIONS {
			"LOGOUT"
				-> "HOME"
		}
	}
	"PATIENT_LOGGED_IN" {
		NAVIGATIONS {
			"LOGOUT"
				-> "HOME"
		}
	}
	"REGISTER" {
		NAVIGATIONS {
			"GOTO_LOGIN"
				-> "HOME"
		}
	}
}
```

References

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

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

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

[4] Whalen, Michael W.; Rajan, Ajitha; Heimdahl, Mats P.E.; Miller, Steven P.:Coverage Metrics for Requirements-based Testing, Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 25-36, ACM New York

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

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

[7] J. DeMott, C. Miller, A. Takanen, "Fuzzing for software security testing and quality assurance," Artech House (2008)

[8] M. Sutton, A. Greene, P. Amini: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley (2007)

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

[10] 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]

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

[12] 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]

[13] 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]

[14] 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]

[15] 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]

[16] 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]

[17] 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]

[18] 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]

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

[20] 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]

[21] 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]

[22] 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]

[23] 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]

[24] MITRE: Common Weakness Enumeration – CWE-294: Authentication Bypass by Capture-replay (2014). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/294.html [Accessed 11 September 2014]

[25] 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]

[26] 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]

[27] MITRE: Common Weakness Enumeration – CWE-352: Cross-Site Request Forgery (CSRF) (2014). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/352.html [Accessed 11 September 2014]

[28] 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]

[29] 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]

[30] 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]

[31] 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]

[32] 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]

[33] 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]

[34] 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]

[35] 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]

[36] 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]

[37] 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]

[38] Open Web Application Security Project: Top 10 2013-A8-Cross-Site Request Forgery (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_%28CSRF%29 [Accessed 11 September 2014]

[39] 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]

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

[41] B. Calin (2012): The Email that Hacks You. [ONLINE] Available at: http://www.acunetix.com/blog/web-security-zone/the-email-that-hacks-you/ [Accessed 19 September 2014]

[42] T. Wilson: Hacker Steals Data on 18M Auction Customers in South Korea. [ONLINE] Available at: http://www.darkreading.com/attacks-breaches/hacker-steals-data-on-18m-auction-customers-in-south-korea/d/d-id/1129325? [Accessed 19 September 2014]

[43] Fraunhofer FOKUS: Fuzzing library Fuzzino on Github (2013). [ONLINE] Available at: https://github.com/fraunhoferfokus/Fuzzino[Accessed 19 September 2014]