A Trace Management Platform for Risk-Based Security Testing

Juergen Grossmann[1], Michael Berger[1], and Johannes Viehmann[1]

Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, D-10589 Berlin,
`juergen.grossmann | michael.berger`
`| johannes.viehmann@fokus.fraunhofer.de`

Abstract The goal of risk-based security testing is to improve the security testing process in order to cover especially risky areas of the application under test and at the same time minimize the time to market and to improve the use of resources by focusing testing work on areas with the highest risks. In RBST risk factors are identified and risk-based security test cases are created and prioritized according to an applicable selection strategy. One of the challenges in RBST is to keep track of the different artifacts that are often managed by different tools. Traceability is the key to manage complex systems in development and testing. This paper introduces *RISKTest*, a trace management platform on the basis of Eclipse that supports the creation and documentation of cross-tool relations during test development and test execution. *RISKTest* is dedicated to risk-based security testing. Thus, we concentrate on the management of traces between the artifacts from risk assessment and testing and the definitions of services that automatically analyze the related artifacts for security and testing related aspects. *RISKTest* has been developed in the DIAMONDS project and evaluated within the project's case studies.

1 Introduction

Today, increasingly complex systems are developed. Several developers create different parts of models and artifacts that represent the system under development. Each developer has specific views on the system with respect to his role in the development process. For instance, the requirement engineer develops the requirement model while the tester creates a test model on basis of the requirements and the system model. Test runs and test results are often summarized by means of a test management tool. There are specific constraints for creating and visualizing the artifacts that are created and managed by different tools. For instance, a test management tool provides an overview of test cases and test runs e.g. in a tree hierarchy. The test results like verdict information are attached in a way that a manager can easily derive the current state of the test process. In contrast, a requirement tool will manage the requirements in a table format that allows to manage the requirements in a hierarchically structure. Therefore the requirements are arranged in rows and the requirement parameters like description, links or technical aspects in columns. Both tools and perspectives are necessary and important. However, to assess the coverage between requirements and tests or test results, artifacts from both tools need to be set in relation. For this, the concept of cross tool traceability has been developed [11,1]. Traceability in general defines relationships between different artifacts or models. Such a relationship consists of at least a tuple of elements and is called trace. For example, a trace can refer to a test case in a test model and a requirement in a requirements model, meaning that the test case validates the realization of the requirement. Also, we can distinguish between typed and untyped traceability approaches. While untyped traceability allows the creation of arbitrary traces between every kind of element, typed traceability requires the definition of a trace metamodel to restrict specific traces to specific element types. Untyped traceability approaches are easier to realize but lack information for a detailed analysis of the underlying traceability graph. Typed traceability provides a richer set of information (i.e. the type of a trace and the types of the elements that are in the trace). This extra information helps to distinguish different kind of relationships (e.g. the relationship between requirements and tests from the relationship between requirements and the code or the system model) and thus provides a stronger basis for correctness checks and the analysis of the traceability graph.

All traces constitute the trace model that can be used by analytic tools to evaluate such relationships. Such analytic tools can traverse not only a single model but several models that are connected via traces, also transitively through different models. Cross tool traceability emphasizes the fact that the artifacts are managed by different development tools. A traceability platform, especially a cross-tool traceability platform must meet a set of requirements in order to enable the efficient use by developers. This article introduces *RISKTest*, a plattform for risk-based security testing that provides extensive support for cross-tool traceability between risk assessment artifacts, requirements, test cases and test results. Section 2 motivates our platform and Section 3 provides an overview on our approach to risk-based security testing, which defines the application context for *RISKTest*. Section 4 introduces *RISKTest*, Section 5 describes its application to cases studies and Section 6 summarizes the paper.

2 Motivation

Traceability was originally established for requirements engineering, i.e. Doors with Rhapsody gateway [5] and Reqtify[1]. These trace tools relate requirements from different sources to the implementation code. The goal is to validate the compliance of a system to a set of requirements, to check if only the required functionality is realized and to be able to make impact analysis if requirements are changed. In software development, traceability is mostly used in the area of safety critical systems [7] [9]. In this area safety requirements will be traced with software requirements and the code. So called slices can be used to filter out not necessary parts to verify relevant fragments. In the area of security,

[1] http://www.3ds.com/products-services/catia/portfolio/geensoft/reqtify/

traceability tools are very rare and mostly in a research state. For example in the JESSIE project [2] a tool for security assurance has been developed that provides support for traceability. Traces are created between the (security) requirements, UML models of the security protocols and the respective implementation. The traces are used to generate a dynamic monitoring component for monitoring the software during run-time. Like in the JESSIE project, the focus of most research projects is the methodology of the handling of the trace model (in JESSIE the dynamic monitor generation), but the trace management itself is only handled parenthetically.

Traceability tools are often realized as separate tools that import the information to be traced. One of the largest problems is the maintenance of the trace model during product development. Often, the establishment of trace information is done in the latest development steps with high effort. Online synchronization, that allow the continuous update of changes in the underlying artifacts is often missing. Moreover, these tools often use their own specific visualization of the imported artifacts, which are different from the original tools and thus unfamiliar to the developer.

Since we think, that one of the main features of a traceability platform is to continuously support the developer in creating and navigating traces while developing the elements to be traced, we prefer a traceability solution with directly integration in the development tools. Such a traceability platform must provide an interface to support the creation and deletion of traces and the navigation along selected traces, and it must provide a set of services and analysis functions to fulfill the analysis requirements of the respective development and quality assurance tasks. Also, it must handle the technical gap between the different tools in use. Preferably, the trace creation and navigation is integrated in the development tools directly.

In the ITEA project DIAMONDS [3] we have introduced the idea of traceability to support risk-based security testing. The basic idea behind risk-based security testing is to use artifacts from the risk-assessment to support the security testing process. Currently there is no method or framework that allows for the systematically capture security risks (i.e. threat scenarios, vulnerabilities, countermeasures) and risk values and relate them with testing artifacts so that test identification and test selection is effectively supported. We are interested in establishing and documenting traces from the risk assessment to the testing artifacts. These traces need to be persistent and operational so that we can navigate along the traces and use the traces as basis for our test evaluation, e.g for calculating the coverage of risk assessment artifacts by tests and test results. Considering this background, the key requirements for our traceability platform are:

– The trace management (the creation, deletion of traces) is directly integrated in each of the development tools. This becomes necessary to support rapid, convenient and continuous usage of the traceability functions while creating the development artifacts (e.g. the risk assessment and security testing artifacts).

– The traceability platform allows for bidirectional navigation between related elements. Navigation source and target should be visualized directly within and by means of the original development tools.
– The creation of traces can be done manually or automatically. The latter is needed to effectively integrate the platform in model-based test generation approaches.
– The traces are defined on basis of a trace metamodel that distinguishes the individual elements that are part of a trace and allows for distinguishing different trace types.
– The trace metamodel defines a service interface that allows for introducing services that query the trace model for information (e.g. services for coverage analysis or impact analysis).
– The traceability platform is extensible. That is, it provides a well-defined interface to easily integrate other development tools that are based on Java and Eclipse.

In the following we give a more concise overview of our approach to risk-based security testing, which yield the concrete application context for the traceability platform. Afterwards we introduce *RISKTest*, our approach to traceability in the area of risk-based security testing.

3 Risk-based security testing

Risk-based security testing (RBST) can be generally introduced with two different goals in mind. On the one hand risk based-testing approaches can help to optimize the overall test process: First, the results of the risk assessment, i.e. vulnerabilities, threat scenarios and unwanted incidents, are used to guide the test identification and may complement requirements engineering results with systematic information concerning threats and vulnerabilities of a system. A comprehensive risk assessment additionally introduces the notion of risk values, that is the estimation of probabilities and consequences for certain threat scenarios. These risk values can be additionally used to weight threat scenarios and thus help identifying which threat scenarios are more relevant and thus identifying the threat scenarios that are the ones that need to be treated and tested more carefully.

Second, risk-based testing approaches can help to optimize the risk assessment itself. Risk assessment, similar to other development activities that start in the early phases of a development project, are mainly based on assumptions on the system to be developed. Testing is one of the most relevant means to do real experiments on real systems and thus be able to gain empirical evidence on the existence or likelihood of vulnerabilities, the applicability and consequences of threat scenarios and the quality of countermeasures. Thus, a test-based risk assessment makes use of risk-based testing results to gain arguments or evidence for the assumptions that have been made during the initial risk assessment phases. In particular risk based testing may help in

- providing arguments or evidence on functional correctness of countermeasures,
- determining the likelihood of exploiting vulnerabilities as described by threat scenarios will lead to unwanted incidents, and
- discovering unknown risk factors (i.e. new vulnerabilities).

In summary, RBST approaches makes use of risk assessment results to focus, optimize, and prioritize the security testing, and the test-based risk assessment is empirically grounded by security testing results. To this end we have identified three distinct activities that constitutes our basis of a risk-based security testing approaches:

- **Risk-based security test planning:** The goal of risk-based security test planning is to improve the testing process systematically: High-risk areas of the application under test can be covered and in same time it can achieve a reduction in the expenses and the resources used by the test work. This will be focused on areas with the highest risks. Moreover, selected test strategies and approaches are identified to address the most critical vulnerabilities.
- **Risk-based security test identification and selection:** Finding an optimal set of security test cases requires an appropriate selection strategy. Such a strategy takes the available test budget into account and also provides, as far as possible, the necessary test coverage. In functional testing, coverage is often described by the coverage of requirements or the coverage of model elements such as states, transitions or decisions. In risk-based testing we aim for the coverage of identified risks of a system. Risk-based security test selection criteria can be used to control the selection or the selected generation of test cases. The criteria are designed by taking the risk values from the risk assessment to set priorities for the test case generation as well as for the order of test execution.
- **Security risk control:** The decision how extensive testing should be is always a question of the remaining test budget, the remaining time and the probability to discover even more critical errors, vulnerabilities or design flaws. In RBST risk analysis gives a good guidance where to find critical errors and which kind of risks have to be addressed (see above). On the other hand, the test results can be used to verify the assumptions that have been made during risk analysis. Newly discovered flaws or vulnerabilities need to be integrated in the risk analysis. The number of errors in the implementation of a countermeasure hints at the maturity of it and allows to assess their adequacy in the security context. In order to allow such an assessment, a sufficient degree of test coverage is required. In this sense, the test results can be used to adjust risk assessment results by introducing new or revised vulnerabilities or revised risk estimations based on the errors or flaws that have been found. Test results, test coverage information and a revised or affirmed risk assessment may provide a solid argument that can be used to effectively verify the level of security of a system.

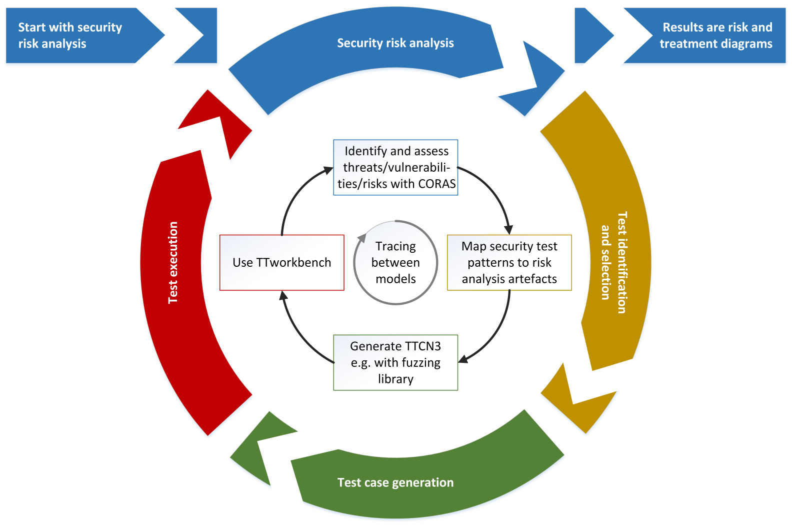

Start with security risk analysis
Security risk analysis
Results are risk and treatment diagrams
Test identification and selection
Test case generation
Test execution
Identify and assess threats/vulnerabili-ties/risks with CORAS
Map security test patterns to risk analysis artefacts
Generate TTCN3 e.g. with fuzzing library
Use TTworkbench
Tracing between models

Figure 1. Risk-based security testing approach

In the following, we outline our approach to RBST. The approach has been developed in the DIAMONDS project [3] and the development is currently continued in the RASEN project [12]. Figure 1 shows the overall interaction between risk analysis and testing depicting both approaches, the optimization of the testing approach by means of risk assessment results and the control and optimization of the risk assessment results by means of test results.

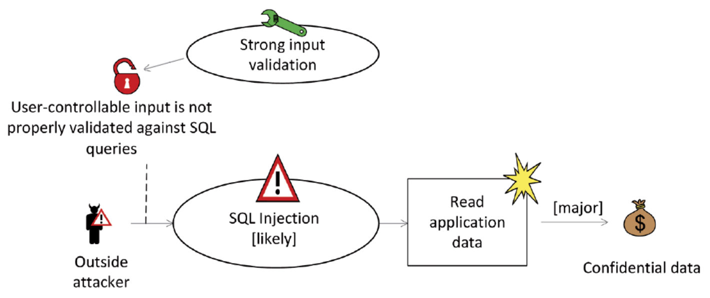


Figure 2. Excerpt of a CORAS-risk-analysis

– **From risk assessment artifacts to test pattern:** Security risk assessment is conducted using CORAS [8]. The CORAS method is performed until we have a first risk estimation. The results of this analysis are expressed by means of threat diagrams containing a qualified description of potential vulnerabilities, threat scenarios and unwanted incidents annotated with likelihood and consequence values (see Figure 2). This initial analysis is based on literature, vulnerability databases and the system model. Its results are highly dependent on the experience and the skills of the risk analysis team. Important aspects might have been missed completely and the just guessed likelihood values are eventually very uncertain.

– **Selecting elements to test:** While the threat diagram immediately can be interpreted as a guide telling the analysts what should be tested, it is not obvious which tests are the most critical and which tests will probably not have a significant impact on the overall risk picture. Since, security testing can be expensive and since, often both time and resources available for testing are rather limited, it would be most helpful to identify the most relevant test cases and to test these in the first place. Within our approach, the CORAS threat diagram is used to identify nothing but the most critical threat scenario that has not yet been tested.
– **Map security test patterns to threat scenarios:** Knowing what should be tested next is fine. However, it can be challenging to create effective test cases and create appropriate metrics that allow sound conclusions for the likelihood values in the threat diagrams. It makes sense to create and to use a catalogue of test patterns [13]. Security test patterns typically consist at least of a name, a context, a problem and a solution description. In our approach, the threat scenario is a direct counterpart to the problem description of a test pattern. Hence, it is easy to identify a fitting test pattern for the most critical threat scenario that needs to be tested if such a pattern already exists in some database. Eventually the same test pattern will have to be instantiated multiple times for a single threat scenario because there are different vulnerabilities and unwanted incidents it can be mapped to.
– **From test patterns to test implementation and execution:** A test pattern contains at least a brief description how the test should be implemented and thereby eventually helps to prevent some potential implementation errors. Test patterns can contain generic template code that can be instantiated into executable code. Our specific model-based security testing methodology uses TTCN-3 [4] to describe the test cases in a flexible and implementation independent way. Test patterns can be described in a flexible way using TTCN-3 notation [15]. For effective security testing, often lots of different test cases have to be generated that are close to the limits of valid input sequences. Typically, in such a case not all test cases are created manually. Instead, model based fuzzing can be used to generate appropriate random test sequences. There is already tool support available for automatic test-case generation producing TTCN-3 code, e.g. for model-based fuzz testing the library developed within the DIAMONDS project [3]. To apply the test cases once TTCN-3 code is generated, any compatible test execution environment may be used. Within the DIAMONDS project, the commercial TTworkbench from Testing Technologies has been used successfully as the

test development and execution environment for the model-based security testing methodology described here.
- **From test results back to risk assessment:** The test results are summarized and evaluated. For each vulnerability we clearly discover the number of executed test cases and their test verdicts. On the basis of these results, the testing as well as the risk assessment is adjusted. If the test results are unclear or the number of the test cases and thus the coverage is insufficient to make a clear security statement, further tests may be initiated. If the tests allow for a security statement the likelihood values in the risk assessment can be adjusted if necessary. In case, new vulnerabilities are discovered by testing, they have to be additionally integrated into the risk assessment.

4 *RISKTest* trace management platform

The *RISKTest* trace management platform is based on a provisional version of the trace management tool CReMa [14] developed by Itemis in the research project VERDE. It is integrated in the desktop development environment of the Eclipse workbench and runs with the modeling tools Eclipse configuration in the versions JUNO and INDIGO. The trace management capabilities, i.e. the creation of trace links, the navigation of trace links and the evaluation trace links, are restricted to a set of integrated tools. These tools are the risk modeling tool CORAS, the Eclipse UML modeling editor Papyrus, the requirement modeling tool ProR, based on the ReqIf model, and the test managing tool TTworkbench. Additionally, interfaces to integrate other modeling tools are developed.

- CORAS has been used for security risk assessment,
- ProR has been used for security requirements engineering and as a data base for the security test pattern catalog,
- Papyrus has been used for security test specification and modeling, and
- TTworkbench has been used for security test execution.

4.1 Domain model abstraction

For a risk driven traceability tool the bulk of tools are the most important part of interest. Each tool has to be integrated into the tool landscape and the interaction between such tools has to be specified. We need tools for risk assessment, for test definition and also for system modeling. We do not want to create a single tool for all modeling parts but want to use specific tools for each part of risk driven development to get the benefit of each of specified tool. We defined different domains the tools are related to: the risk domain for developing risk assessment models, the test domain for test cases simulating attacks on detected vulnerabilities, the system domain for specifying system components and interface interactions between them, and as the fourth domain the requirement domain to specify requirements for test cases and for a test pattern catalog. To synchronize each tool in a domain, a domain-metamodel is specified on basis of some general terms and artifacts given by standards. The table below provide the basic terms for the domains of risk assessment and testing.

Artifact	Description
Security Risk	Security risk is a risk caused by a threat of exploiting a vulnerability and thereby violating a security requirement.
Unwanted Incident	Unwanted incident is an event representing a security risk.
Threat	Threat is potential cause of an unwanted incident [10].
Vulnerability	Vulnerability is weakness of an asset or control that can be exploited by a threat [10].
Test Case	A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement [6]
Test Result	The consequence/outcome of the execution of a test. It includes outputs to screens, changes to data, reports and communication messages sent out [6].
Test Pattern	An artifact that specifies a set of best practices to achieve dedicated test objectives in the context of a certain testing problem. Just as design patterns capture design knowledge into a reusable medium, test patterns capture testing knowledge into a reusable medium.

These terms and artifacts form the basis of our domain specific models. Traces and services are working with the domain elements instead of the elements that come from the data models of the concrete development tool. This introduces a level of abstraction that eases the integration of new development tools. For this purpose only a mapping between the development tool and the domain metamodel has to specified. The interactions of the different domains are already given by the trace-metamodel and thus managed by the trace management framework. Figure 3 shows the trace model that has been used as basis to enable the traceability support for risk-based security testing.

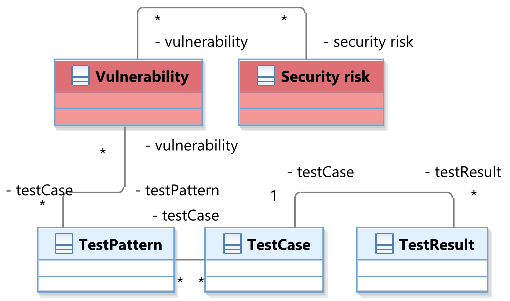


Figure 3. Trace metamodel for the testing and risk domain

4.2 Improved user interaction directly from within the tools

To support the user with an easy interaction we integrated the trace administration directly in the model development editor views. The user can

- create new traces between selected elements in all supported editors,
- navigate to a traced element from a selected element,
- delete a trace between the selected element and a traced element, and
- edit one of the traces of a selected element.

The main advantage over most other traceability tools is the complete integration of the interaction triggers in the user interfaces of original tools, so that the user can develop the model and define traces with the same tool interface.

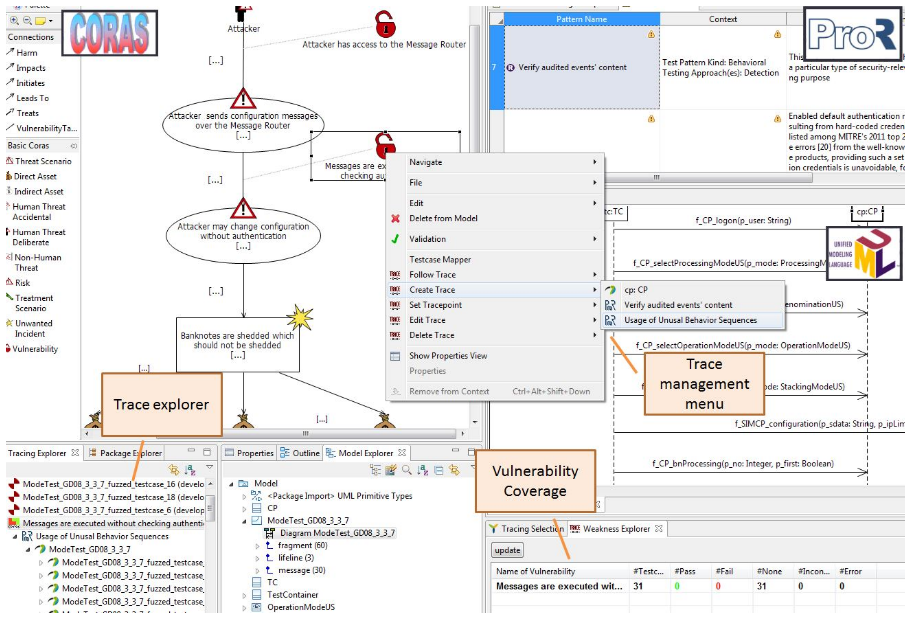


Figure 4. Traceability management embedded in Eclipse (based on CReMa)

All traced elements can be administrated with the trace explorer. The explorer visualizes all traces and enables navigating through the trace model and focusing on traced elements using the corresponding editor view. Also, a filter mechanism is provided to hide non-relevant element types. Figure 4 shows an exemplary screenshot of our current implementation.

4.3 High level architecture

The architecture for *RISKTest* can be divided into three layers: The service layer contains all services operating on the models. The traceability layer constitutes the core of the trace management tool and implements query handling. The domain layer contains all domains and editor tools including their metamodels (see Figure 5). In the following we describe the concept of our trace management framework layer by layer and component by component.

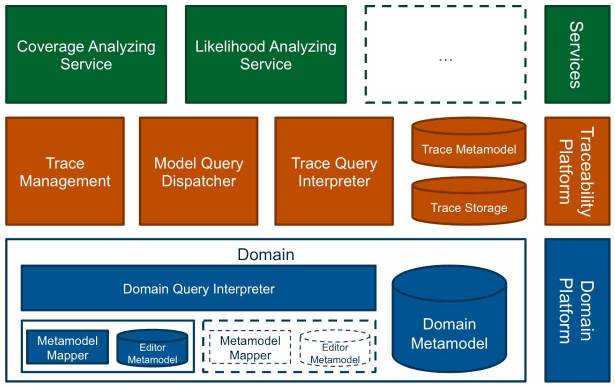


Figure 5. Trace management framework in multi-layer diagram

- **Traceability Platform Layer**: All trace handling components are specified in the middle layer. These components manage the creation and modification of traces, use the information of the trace metamodel and process query requests.
- **Domain Platform Layer:** In order to enable the development of services for the traceability managing tool, it is necessary to provide a unified metamodel of each domain. This unified domain metamodel allows services to access elements from editors using different metamodels. Therefore, it is necessary that the editor dependent metamodel is mapped to the unified domain metamodel. This is done with an editor-specific extension to the domain platform layer.
- **Service Layer:** The whole trace management framework is useless without the services. Each service uses parts of information from the models. By using the domain metamodels and the trace metamodel, queries will be defined to get this information. The queries are sent from the services to the query dispatcher and answered with a set of elements and relations. A service is triggered by traceability management component e.g. when a user invokes corresponding functionality within an editor. For example, a service has to analyse the coverage of test cases and the related system components. The results may then be highlighted within the editor.

The *Traceability Platform Layer* consists of a number of components for creating and analyzing the trace links:

– **Traceability Management:** All user interactions for editing, viewing and deleting traces are implemented in the trace management component. The traceability editing functions consists of a trace editor and the trace explorer. The trace editor enables setting the trace type, assigning elements to a trace selected within a model editor, and obtaining other trace information like source model or name of each involved element. In the trace explorer all traces are shown. It allows navigating and creating filter to hide non-relevant traces.
– **Trace Metamodel:** The trace metamodel defines the different types of traces and specifies between which kind of elements these traces can be created. The traceability management uses the trace metamodel in order to constrain the creation of traces. The trace types as well as domain-specific elements, their attributes and relationships within a domain can be referenced within a query. The query dispatcher uses the trace metamodel to manage the query processing. In order to support new services, it is easy to extend the trace metamodel.
– **Trace Storage:** The trace storage stores all traces. This store may be located on a local disk. While this is sufficient in a single-user environment, teams require to access and modify traces from different workstations. For that purpose, the trace metamodel can be stored in a network repository and version control system. The project EMFStore provides such a distributed storage.
– **Model Query Dispatcher:** In this component the strategy to collect information from the models depending on a querying service is specified. The dispatcher service distributes queries or parts of queries to the different query interpreter components and collects the result information.
– **Trace Query Interpreter:** The task of the trace query interpreter it to solve the submitted queries. The trace query interpreter can only resolve requests specified for the trace model but cannot process queries specified on elements without a trace relation in the trace meta-model.

The *Domain Platform Layer* allows services to access elements from the individual tools or editors by mapping the unified domain metamodel to the editor models of the tools:

– **Domain Metamodel:** The domain metamodel is an abstract model and defines the artifacts and their relations for a certain domain. Since all framework services are based on the domain metamodels without knowledge of the underlying tools, the domain metamodel must be rich enough to provide all the information that are needed by the services.
– **Domain Query Interpreter:** The domain query interpreter is comparable to the trace query interpreter with the only difference that it is dedicated to process queries on the level of the domain metamodel and not on the level of the trace metamodel. The domain query interpreter processes information that relate to instances of the domain metamodel and a dedicated mapper can associate the domain elements to the elements of the underlying editor

model. That reduces the complexity of queries by working on domain metamodels that are often less complex than editor metamodels because they are designed to fit only certain use cases.

- **Editor Metamodel:** The editor metamodel is the metamodel that describes the data managed by the individual tools or editors. For example, Papyrus belongs to the system domain and uses UML2 as metamodel. The requirements domain belonging editor ProR uses ReqIF, and the risk model editor CORAS uses the CORAS metamodel. Such models will not be queried directly but by using the mapping to the domain metamodel.
- **Model Mapper:** The model mapper defines a mapping between the domain metamodel and a metamodel of a certain editor.

5 Application to case studies

RISKTest had been used in two case studies. The Giesecke & Devrient (G&D) case study deals with the security testing of a banknote processing machine and the Dornier Consulting case studies deals with the security testing of a head unit from the automotive domain. During the security test development the trace management platform had been used to create and maintain trace links between risk assessment artifacts (i.e. vulnerabilities, threat scenarios, and treatment scenarios), test pattern and test specification. The test developer starts with the risk assessment tool CORAS and identifies security test objectives and security testing approaches by relating test pattern from the test pattern library in ProR to risk assessment results in CORAS. Based on these initial assignments the test developer starts specifying the test cases in Papyrus following the ideas given by the test pattern. Each of the test models are again linked to the corresponding test pattern, so that we get a transitive trace link to our initial test basis (i.e. risk assessment results). As DIAMONDS provides model-based testing approaches we normally use test generators to generate the test cases from the test models. The test generator had been integrated in that way, that it adds and updates traceability links from the test models to the generated test cases. Thus, during the whole security test development process the test developer has full control over all dependencies that are made persistent by means of the trace management platform. The developer can manually navigate along the links and actively switch between the different models, artifacts and perspectives. He can easily control the current status of the test development process by analyzing the coverage of the risk assessment elements with test pattern, test models and test cases.

The testing process in the Giesecke & Devrient case study had started with a concise security risk assessment revealing threat scenarios and associated potential vulnerabilities like, e.g., the “authentication bypass of the Message Router” or the “SQL injection into the database”. The potential vulnerabilities identified during the risk assessment were related with test patterns and the initial test models manually. The *RISKTest* trace management framework supports the user with an easy way to create such manual relationships. Afterwards, a model-based test generation approach had been used to generate a large number of test cases on basis of the initial test model. To be able to capture the relations between the initial test model and the executable tests, we had developed an interface that allows an automatic generation of traces during the test generation process. Finally, the user has the possibility to relate test results automatically to the appropriate test cases so that the complete traces from the initial vulnerabilities to the related test results are defined. *RISKTest* is now able to calculate the coverage of the initial vulnerabilities by associated test results.

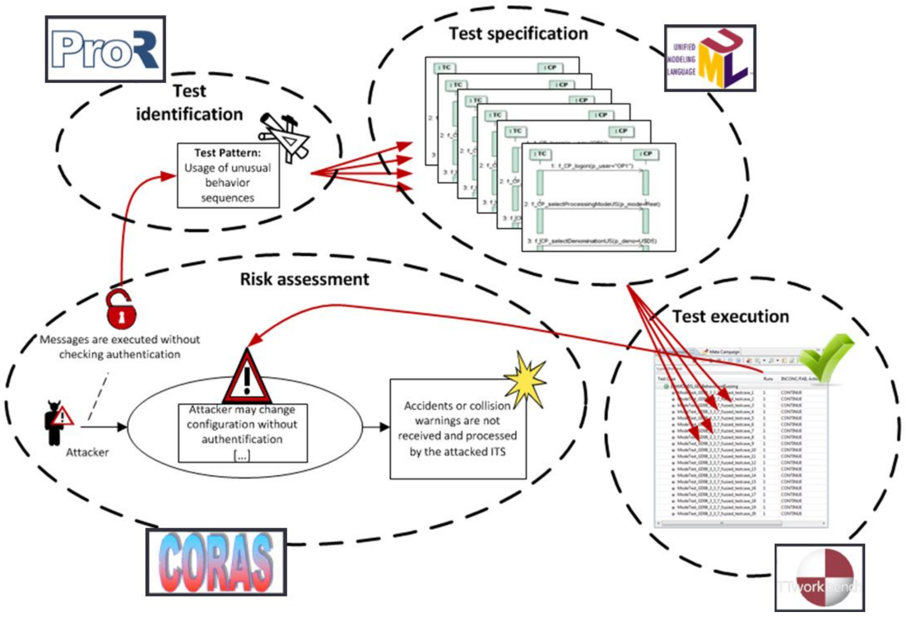

Test identification
Test Pattern: Usage of unusual behavior sequences
Test specification
Risk assessment
Messages are executed without checking authentication
Attacker
Attacker may change configuration without authentification [...]
Accidents or collision warnings are not received and processed by the attacked ITS
Test execution

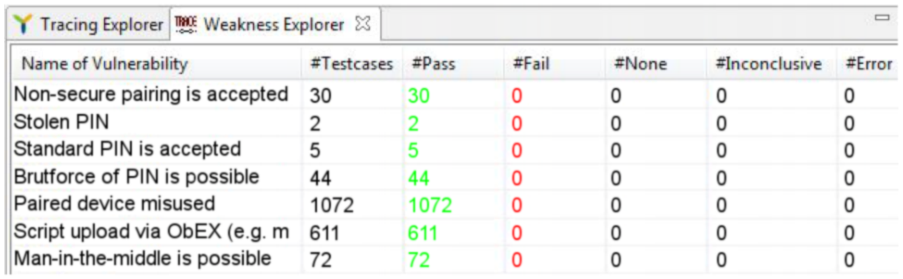
Figure 6. Traceability from risk assessment artifacts to test results

Figure 6 provides an overview over the classes of artifacts that are to be related. The red arrows visualize traceability links. The dashed shapes represent the scopes of the different tools.

Based on the risk models, 30 behavioural fuzz test cases were executed on the SUT regarding an authentication bypass. Additionally, an initial set of 24 test cases using SQL injection to bypass the authentication were executed. No security-related issues were found. The *RISKTest* framework provides a view that shows the test coverage of selected vulnerabilities and updates them on basis of the traceability information in the *RISKTest* framework (See below).

While the initial traces between vulnerabilities from the risk model, the behavioural model of the SUT and the chosen security test patterns have to be created manually, most traces that results from test case generation and execution are generated automatically. This allows a semi-automatic measurement of risk coverage.

The second use case gives attention to the test documentation and test result aggregation. The use case partner Dornier executed 1836 test cases with an test manager developed by dornier itself. Because the test tool was not based on eclipse, it was not integrated in our trace framework. But the test documentation and test results are exported as XML files and imported into the framework. The test results are automatically traced to the related test pattern or directly to the analyzed vulnerabilities. Based on the traces we were able to calculate the coverage of vulnerabilities by security test cases. Figure 7 shows the aggregated test results for a set of vulnerabilities from the Dornier case study. It shows in each line for a vulnerability from the risk model the test verdict of the test cases that are linked back to the vulnerabilities from the risk model. All vulnerabilities with direct or transitive relations to test cases are listed and shown with their related test cases and test results. Measured by means of the vulnerability coverage, we could detect the coverage over all risks and that no risk was untested.



Name of Vulnerability	#Testcases	#Pass	#Fail	#None	#Inconclusive	#Error
Non-secure pairing is accepted	30	30	0	0	0	0
Stolen PIN	2	2	0	0	0	0
Standard PIN is accepted	5	5	0	0	0	0
Brutforce of PIN is possible	44	44	0	0	0	0
Paired device misused	1072	1072	0	0	0	0
Script upload via ObEX (e.g. m	611	611	0	0	0	0
Man-in-the-middle is possible	72	72	0	0	0	0

Figure 7. Potential vulnerabilities and related tests and test results

6 Summary and outlook

The introduced combination of risk analysis and security testing shows a high potential to improve the systematic quality assurance of security critical systems. On the one hand, risk analysis provides a proper guidance for a systematic test identification and test prioritization. On the other hand, security testing and the analysis of security testing results can provide evidence on assumptions that have been made during risk analysis. With sufficient tool support, traceability between risk analysis artifacts and testing artifacts can be operationalized and monitored during the system development. This paper has introduced *RISKTest*, a trace management platform that supports the main activities of risk-based security testing. It supports the development of security test cases and helps understanding the test results by reflecting their relationship to risk assessment. The *RISKTest* framework is set up in a way that it completely integrates with the development tools and allows the security test developer to create and manage cross-tool relations and traces directly from within the original development tools. For the next iteration the framework is planned to improve the capabilities of the query interface and enhance and to develop dedicated algorithm for risk-based test selection and prioritization. Further development of the framework will be done in the FP7 project RASEN [12].

References

[1] Altheide, Frank ; Schuerr, Andy ; Doerr, Dr. H.: Requirements to a Framework for Sustainable Integration of System Development Tools. In: *Proc. of the 3rd European Systems Engineering Conference (EuSEC)*, 2002, S. 53–57

[2] Andreas Bauer, Jan J. ; Yu, Yijun: Run-Time Security Traceability for Evolving Systems. 2008. – Forschungsbericht

[3] DIAMONDS: *Website of the ITEA project DIAMONDS (Development and Industrial Application of Multi-Domain-Security Testing Technologies)*. http://www.itea2-diamonds.org/, 2013

[4] ETSI: *Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language (ETSI Std. ES 201 873-1 V4.3.1)*. Sophia Antipolis, France, Febr. 2011

[5] IBM Corporation: *IBM Rational Rhapsody Gateway Add on User Manual*. 2001-2010

[6] ISTQB: *ISTQB Glossary of testing terms version 2.2*. http://www.istqb.org/downloads/finish/20/101.html, 2013

[7] Katta, V. ; Stalhane: A Conceptual Model of Traceability for Safety Systems

[8] Lund, M. ; Solhaug, B ; StÃ¸len, K.: *The CORAS Approach*. 1st Edition. Springer-Verlag, 2011 (ISBN 978-3-642-12322-1)

[9] Nejati, Shiva ; Sabetzadeh, Mehrdad ; Falessi, Davide ; Briand, Lionel ; Coq, Thierry: A SysML-Based Approach to Traceability Management and Design Slicing in Support of Safety Certification: Framework, Tool Support, and Case Studies / Simula Research Lab. 2011 (2011-01). – Forschungsbericht

[10] Organization, International S.: *ISO 27000:2009 (E), Information technology - Security techniques - Information security management systems - Overview and vocabulary*. 2009

[11] R, Freude ; A., Koenigs: Tool integration with consistency relations and their visualization. Helsinki. In: *Proceedings of the ESEC/FSE 2003 workshop on tool integration in system development, Helsinki*, 2003

[12] RASEN: *Website of the FP7 project RASEN (Compositional Risk Assessment and Security Testing of Networked Systems)*. http://www.rasen-project.eu/, 2013

[13] Smith, Ben H. ; Williams, Laurie: On the Effective Use of Security Test Patterns. In: *SERE*, IEEE, 2012. – ISBN 978–0–7695–4742–8, S. 108–117

[14] VERDE: *Yakindu CReMa - the nice thing on top of eclipse*. http://www.guersoy.net/knowledge/crema, 2011

[15] Vouffo-Feudjio, Alain ; Schieferdecker, Ina: Test Patterns with TTCN-3. In: Grabowski, Jens (Hrsg.) ; Nielsen, Brian (Hrsg.): *FATES* Bd. 3395, Springer, 2004. – ISBN 3–540–25109–X, S. 170–179