Combining Risk Analysis and Security Testing[1]

Jürgen Großmann, Martin Schneider, Johannes Viehmann, Marc-Florian Wendland

Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
D-10589 Berlin
Germany
{Juergen.Grossmann, Martin.Schneider, Johannes.Viehmann, Marc-Florian.Wendland}@Fokus.Fraunhofer.de

Abstract.
A systematic integration of risk analysis and security testing allows for optimizing the test process as well as the risk assessment itself. The result of the risk assessment, i.e. the identified vulnerabilities, threat scenarios and unwanted incidents, can be used to guide the test identification and may complement requirements engineering results with systematic information concerning the threats and vulnerabilities of a system and their probabilities and consequences. This information can be used to weight threat scenarios and thus help identifying the ones that need to be treated and tested more carefully. On the other side, risk-based testing approaches can help to optimize the risk assessment itself by gaining empirical knowledge on the existence of vulnerabilities, the applicability and consequences of threat scenarios and the quality of countermeasures. This paper outlines a tool-based approach for risk-based security testing that combines the notion of risk-assessment with a pattern-based approach for automatic test generation relying on test directives and strategies and shows how results from the testing are systematically fed back into the risk assessment.

Keywords: Risk assessment, security testing, test pattern

1 Introduction

Security is crucial in various market sectors, including IT, health, aviation and aerospace. In the real world perfect security often cannot be achieved. Trust allows human beings to take remaining risks. Before trusting, before taking risks, it is reasonable to carefully analyze the chances, the potential benefits and the potential losses as far as possible. For technical systems, services and applications such an analysis might include risk assessment and security testing.

[1] The research leading to these results has also received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreements no 316853 and no 318786.

Those offering security critical technical systems, applications or services can benefit from careful risk analysis and security testing in two ways: They can use the results to detect and treat potential weaknesses in their products. Additionally, they can use the results to communicate the identified remaining risks honestly, which can be very important to create trust.

This paper introduces new concepts to integrate compositional risk assessment and security testing into a single process. Furthermore, ideas for increasing the reusability of the risk analysis and security testing artifacts are presented. A focal point is laid onto the systematic integration of risk information and the test design process. In this paper, we present an approach to model-based risk-driven test design. We are going to employ so called test models, a formal specification of test design techniques (as part of the model) as well as model-based representation of risk information in order to actually generate test artifacts (such as test cases and test data) based on risk.

Implementing the described methodology in a tool in order to make it practically applicable for large scale systems for which manual analysis is not practicable is currently ongoing work.

2 The Problems

There is little doubt that security critical technical systems should be carefully analyzed. However, both risk assessment and security testing might be difficult and expensive. Each activity on its own requires experts. The systematic integration of both is a non-trivial challenge, especially in industrial projects where testers are not necessarily experts in security testing and risk analysis.

Typically risk assessment is performed at a high level of abstraction and results depend on the experience and on subjective judgment of the analysts. Hence, results might be imprecise, unreliable and uncertain.

In contrast to risk assessment, security testing does produce objective and precise results – but only for those things that are actually tested. Even for small systems, complete testing is usually not possible since it would take too long and it would be by far too expensive. The selection of relevant test cases is a critical decision. Even highly insecure system can produce lots of correct test verdicts if the “wrong” test cases have been created and executed. Thus, the selection of appropriate test cases needs to be carried out in a systematic and comprehensible manner. Risk-based testing in general is a means to justify why a certain test case or test datum has been designed and executed in the first place. According to Bach [15], risk-based testing aims at testing the right things of a system at the right time. The idea of risk-based testing is simple: Identify prior to test case design those scenarios that provoke the most critical situations for a system to be tested and ensure that these critical situations are both effectively mitigated and sufficiently tested.

Additionally, test results can be systematically integrated with the risk assessment results in order to verify whether the assumed criticality of the system to be tested actually corresponds to the actual implementation of the system. By doing so the risk analysis results can be refined with respect to the actual test results. If the tests pass, the security properties are met and the integration of the security measures seem to be properly realized. Thus, we have gained confidence that at least for the tested situations the system is secure. If a test has failed, thus if an unwanted incident occurred, we gained confidence that the countermeasures are insufficient or could be circumvented for at least one case. This might require additional counter measures to be specified and implemented or deficient countermeasure implementations to be fixed.The main problem is that there is a lack of reliable and applicable methodologies on how to integrate the various pieces of information relevant for a risk-based testing or test-based risk assessment approach in a systematic way.

3 State of the Art

Risk assessment means to identify, analyze and evaluate risks which threaten assets [1] [2]. There are lots of different methods and technologies established for risk assessment, including fault tree analysis (FTA) [4], event tree analysis ETA [5], Failure Mode Effect (and Criticality) Analysis FMEA/FMECA [3] and the CORAS method [6]. However, most traditional risk assessment technologies analyze systems as a whole [7]. They do not offer support for compositional risk assessment. Compositional risk assessment combines risk analysis results for components of a complex modular system to derive a risk picture for the entire complex system without looking further into the details of its components. Nevertheless, for the mentioned risk assessment concepts, there are some publications dealing with compositional risk analysis, e.g. [8] for FTA and [9] for FMEA.

While traditional testing tries to test specified functionality, security testing aims for identifying weaknesses and vulnerabilities to uncover unwanted behavior.

Currently, there are basically two different ways how security testing and security risk analysis can be combined [10]. Test Based Security Risk Assessment (TBRA) tries to improve the security risk analysis with the help of security risk testing and the final output results are risk analysis artifacts. There have been several publications about this approach, e.g. [11] [12], but there is no general applicable methodology and not much tool support.

In contrast to Test Based Security Risk Assessment, the Risk Based Security Testing (RBST) approach tries to improve security testing with the help of security risk analysis and the final results are test result reports. There are lots of different methods, some trying to identify test cases while others try to prioritize test cases or to do both. For example, [13] uses fault trees as the starting point for identifying test cases. Stallbaum and Metzger automated the generation of risk-based test suites based on previously calculated requirements metrics [16] [17]. A prototype research tool called RiteDAP has been presented as being able to generate test cases out of weighted activity diagrams. Basically it ranks paths in the activity diagram due to the risk they include.

Bauer and Zimmermann have presented a methodology called *sequence-based specification* to express formal requirements as low-level mealy machines for embedded safety-critical systems (e.g., [18] [19]). They build a system model based on the requirements specification. Afterwards, the outcome of a hazard analysis is weaved into the mealy machine. The correctness of the natural language requirements is actually assumed to hold. Finally, they describe an algorithm that derives test models that include critical transitions out of the system model for each single identified hazard in order to verify the implementation of a corresponding safety function.

Chen discussed an approach for risk-based regression testing optimization [20]. In his approach the author applies a risk value to each test case to prioritize them. Based on these risk values, the test cases are comparable and can be prioritized to either be included in, or excluded from, a re-running regression testing process.

Security testing and thus risk-based security testing could additionally gain from reusing existing test knowledge. Security test patterns are a way to formulate a solution for recurring security testing problems in a structured way where the solution of such a pattern is used in a different way each time [23]. Several patterns in the context of security testing were already defined by [23] and [24]. These patterns describe the problem or goal to be solved as well as the solution to solve this problem or to achieve this goal. They also refer to known applications, other test patterns and categorize the kind of pattern along the security approach (e.g. 'prevention' for patterns that impede unwanted incidents). However, all existing patterns have in common that they do not provide the information in a way that allow (semi-)automatic test case generation for the purpose of risk-based security testing.

With RBST and TBRA there are at least two different ways how risk assessment and security testing can interact, but of course it should be possible to combine both, too. [11] uses a combination of both approaches, but it does not propose any technique or detailed guideline for how to update the risk model based on the test results.

In this paper, we describe an approach that will combine RBST, TBRA and the use of security test pattern. The approach will be presented together with a methodology specifying how it should be done in the context of a model-based testing process.

4 Combination of RA and Security Testing

TBSR and RBST can benefit from one another. Indeed it might be helpful to switch multiple times between security testing and security risk analysis because after each round of testing and transferring the test results back into the risk picture, the risk analysis might be more precise and thus allow a better identification and prioritization of the next most critical and relevant test cases. Such an iterative process is not linear, it is an incremental process that can be visualized as a cycle. Fig. 1 illustrates our combined TBSR and RBST process.

Note that in our approach, security risk analysis is seen as both the starting point and the end point for the combined TBSR and RBST process. The main reason for this design decision is that risk analysis might also include aspects that cannot be tested while all test results can be regarded as risk analysis artifacts, too.

Security risk analysis can be conducted with any established method. In our implementation we use the CORAS method, which is at the beginning only performed till CORAS step 6, i.e. risk estimation with threat diagrams. The results of this initial analysis are typically expressed with risk graphs or tables containing likelihood and consequence values. 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.

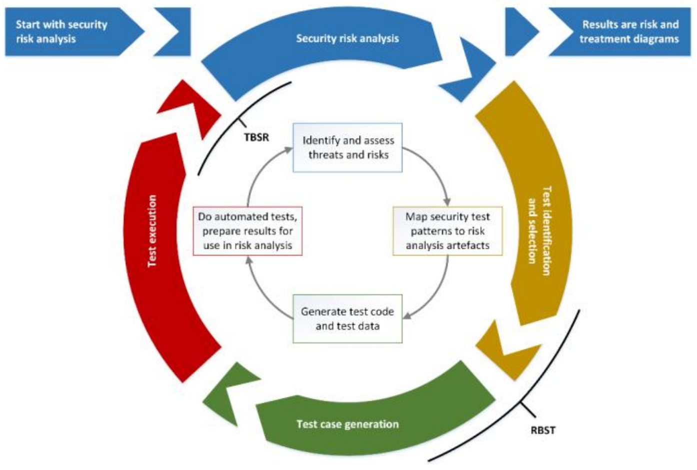

Start with security risk analysis
Security risk analysis
Results are risk and treatment diagrams
TBSR
Identify and assess threats and risks
Test identification and selection
Map security test patterns to risk analysis artefacts
Test execution
Do automated tests, prepare results for use in risk analysis
Generate test code and test data
Test case generation
RBST

Fig. 1. Combined TBSR and RBST process

4.1 Selecting Elements to Test

Though the initial analysis might be imprecise and incomplete, it is a good starting point for the first round of the security testing process, because it gives at least an idea of some things that could go wrong.

While a risk graph like the CORAS threat diagram which contains faults or unwanted incidents can be immediately interpreted as an indicator for what should be tested (i.e. trying to trigger the faults/unwanted incidents), it is not obvious how the testing should be done and which tests are the most significant. 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, at best in an automated way.

Multiple methods can be used to identify the most critical aspects that should be tested in the first place. One risk based prioritization method tries to evaluate the criticality of individual risk analysis artifacts. For example, it can use likelihood values and relations between different elements in a risk graph to calculate likelihood values for dependent incidents or faults. By setting these likelihood values in relation to the potential consequences, for any risk analysis artifact a risk value can be calculated. This approach is appropriate if the highest risks should be tested first.

Another prioritization method is motivated by the fact that risk estimates are often not precisely known. It tries to identify the impact that errors in the estimates for likelihood or consequence values for individual risk analysis artifacts would have for the entire risk picture. Besides assessments by the analysts how much confidence they have in their results and how precise these are, it is also possible to do simulations with the minimal and maximal possible values for each single risk artifact and to compare the resulting overall risk pictures, for example. This method is appropriate to focus on those elements for which the uncertainty of the risk estimation is high and the consequences of errors in the estimates are most significant.

In our combined TBSR and RBST process, only a single risk analysis artifact which should be tested next has to be selected by one of these prioritization methods.

Knowing what should be tested next is fine. However, it can be challenging to create effective test cases and to create appropriate metrics that allow sound conclusions for the risk picture. Instead of reinventing the wheel each and every time, it makes sense to create and to use a catalogue of test patterns. Ideally, the elements of a test pattern library do already contain information how they are associated with certain risk analysis artifacts.

4.2 Applying Test Pattern Using Models

Once all relevant information for the test design process are brought together, consolidated and ready for being exploited by the tester, it needs to be decided how the test design process should be carried out. In our methodology, we rely on a model-based and risk-driven approach for deriving test case, test data and/or test code in an automated, yet comprehensible way. In order to apply model-based techniques a model has to be created or obtained. A model that was designed for the derivation of test artifacts is called *test model.* A test model is a "… model that specifies various testing aspects, such as test objectives, test plans, test architecture, test cases, test data etc." [21]

Models (and so are test models as well) in general are designed for a specific objective. In our methodology, the security test patterns that apply to a certain threat scenario represent these objectives for the design of a test model and, in addition, they specify what test design techniques should be applied to in order to derive test cases. A conceptual model of the dependencies is depicted in Fig. 2. .

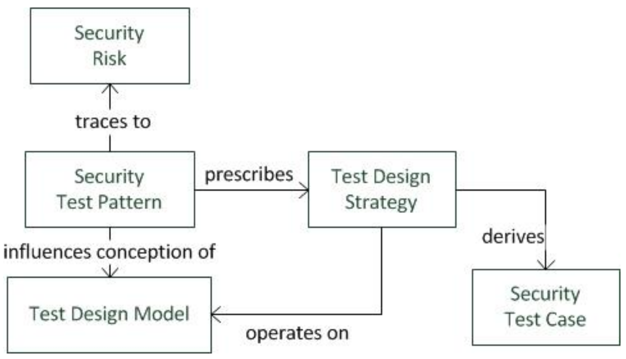


Fig. 2. Testing using patterns

Our approach to model-based test design is based on the UML Testing Profile and an additional extension for describing test design directives and test design techniques.

A *test design strategy* describes a single technique to derive test cases or test data either in an automated manner (i.e., by using a test generator) or manually (i.e., performed by a test designer). A test design strategy represents the logic of a certain test design technique (such as structural coverage criteria or equivalence partitioning) and is understood as logical instructions for the entity that finally carries out the test derivation process. Examples for test design techniques standardized by ISO 29119 [27] are state transitions testing, scenario testing or the data-specific techniques boundary values analysis or equivalence partitioning. Further well-known test design techniques that are frequently applied in model-based testing are described by Utting [22].

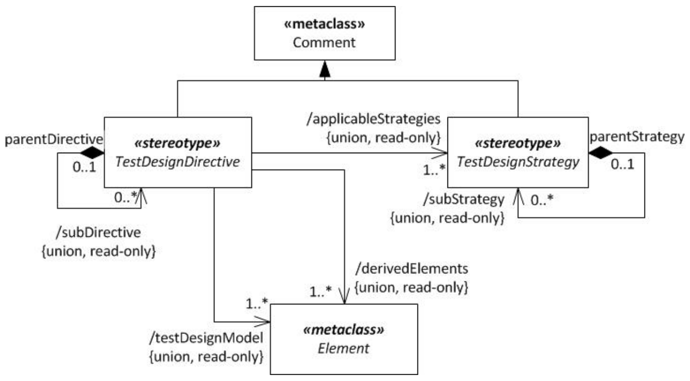

«metaclass» Comment
«stereotype» TestDesignDirective
«stereotype» TestDesignStrategy
«metaclass» Element
parentDirective
0..1
0..*
/subDirective {union, read-only}
/applicableStrategies {union, read-only}
1..*
parentStrategy
0..1
/subStrategy {union, read-only}
0..*
/derivedElements {union, read-only}
1..*
1..*
/testDesignModel {union, read-only}

Fig. 3. UML Profile for Test Design Strategies

A *test design directive* governs an arbitrary number of test design strategies that a test generator has to obey. Therefore, it assembles appropriately deemed test design strategies to eventual fulfill the objective of the security test pattern. In risk-based security testing, we want to ensure that threat scenarios are tested thoroughly because they are deemed critical to the success of the system. In our risk-driven test design methodology, we capture the information on how to derive test cases for a specific risk (transitively referred to through security test pattern) in a test design directive in order to enable an automated derivation of test cases and test data. Directives make the entire test design activities more systematic, understandable and even more important reproducible. The test generation process can be easily adjusted to changed needs by just redefining a test directive's strategy. This means that whenever the risk assessment for a threat scenario is updated, it is easily possible to adjust the intensity of the associated test design strategies and for the respective security test pattern.

The security test patterns so far are defined using natural language. This impedes (semi-) automatic test design. In order to enable test case generation from security test patterns in a (semi-) automated manner, we adapted the structure of security test pattern.

Table 1 shows an excerpt of a security test pattern. The relevant fields for a mapping between risk analysis artifacts and patterns and for test generation are marked gray. An identifier from the Common Attack Pattern Enumeration and Classification [265] allows to map identified threat scenarios from the risk model to security test patterns. In order to support different abstraction levels of risk models where identified threat scenarios are less specific, security test patterns are forming a hierarchy of generalizations. This hierarchy of patterns is realized with the field 'Generalization of'. As a consequence of more general patterns, the test strategies would be more general resulting in a larger number of generated test cases based on such a pattern.

The revised solution of a security test patterns contains beside the solution description in natural language two fields for (semi-)automatic test case generation: test design technique, that identifies a particular method for test case generation, and test strategies, that specifies in which way the test design technique shall be applied in order to generate test cases that are able to find the weakness the security test pattern is intended for. Such test strategies can be implemented by test case generators.

Additionally, the effort of testing for such a vulnerability as well as the effectiveness are recorded for a solution that estimate the effort for testing using the specified solution, i.e. the manual effort, and the effectiveness, i.e. how likely it is to find a vulnerability using the described solution. This information allows to select the best pattern if different patterns are applicable and resources are limited, or to prioritize several patterns. The field 'Metrics' specifies security testing metrics that allow to aggregate test results and estimate the exploitability of a revealed vulnerable based on the test results.

Pattern Name	SQL Injection	
CAPEC-ID(s)	66	
Weakness Description	*Discussion of the weakness in natural language*	
Solution	*Solution in natural language for manual testing*	
	Test Design Technique	Data fuzzing
	Test Strategies	SQL Injection
	Effort	Low to medium: can be highly automated using fuzzing techniques or SQL injection dictionaries.
	Effectiveness	Medium to high, depending on detection capabilities by access to the affected database and to error messages
Metrics	*subject of future research*	
Generalization of	Improper Input Validation	

Table 1. Excerpt of Adapted Security Test Pattern

In order to instantiate such a test pattern, several pieces of information are required. These are at least the interfaces, methods and parameters to be used by a pattern in order to stimulate the system under test. This information can be retrieved from the system under test or from a model of the system under test.

4.3 Test Result Aggregation and Integration

The final step in our combined process (i.e. the actual TBSR step) is the test result aggregation and the integration of the test results into the risk picture. Therefore, the risk analysis process restarts with the test results as new additional input information for the risk analysts. These test results might bring vulnerabilities, threat scenarios and faults / unwanted incidents to attention that were not recognized before.

Additionally, the estimation of likelihoods might become more precise taking the test results into consideration. The risk picture can be iteratively improved by starting again with selecting the next risk analysis artifact that should be studied in detail with the help of security testing.

5 Compositional Risk Analysis and Security Testing Tool

Based on our ideas for RBST in combination with TBSR we have developed a tool providing assistance for the entire process. In order to reduce the amount of manual work as far as possible, the tool tries to maximize the reusability of risk analysis artifacts and testing artifacts and to use automation where it is possible.

For risk analysis, the tool uses an extended version of CORAS supporting compositionality with the help of reusable *threat interfaces* as described in [14]. The risk graph that is generated with our tool contains besides the risk related information some information about the system that is analyzed itself, i.e. the instances of threat interfaces describe the components in detail. This information is valuable especially for automated testing. However, first of all, this system information needs to be introduced into the risk graph. Our tool automatically generates partial *threat interfaces* for components from existing compiled binaries or from source code for the components. Hence, there is no need to manually create models describing the interfaces if none are available.

The risk analysts complete the partial *threat interfaces* by adding unwanted incidents, threat scenarios and vulnerabilities. While this involves some manual work, the analysts can take advantage of our tool's assistants using existing risk related databases like CWE [26] or CAPEC [27]. Inserted risk analysis artifacts can be associated with the system model by drag and drop. For example, CWE based vulnerabilities can be dragged to input ports of threat interface instances, which automatically associates the risk graph element with the system port. The analyst is further supported with suggestions for other nodes like threat scenarios which might typically also be relevant in conjunction with already inserted nodes. For such suggested elements even the relations to present nodes are created automatically as soon as they are inserted.

Using these assistants, negative risk analysis becomes a feasible option for analysts. In negative risk analysis, instead of trying to identify the relevant risks, initially it is assumed that all known risk artifacts (e.g. any CWE and CAPEC derived elements) are relevant. Only those that are for sure not relevant are removed. All remaining risks are considered to be relevant risks until proven otherwise.

Once the *threat interfaces* are complete with all the risks and relations that the analysts can identify without testing or simulation, the next step to verify and improve the risk picture is test identification and selection. Our tool automatically identifies appropriate test patterns for many CAPEC based threat scenarios from its test pattern library. Based on the potential consequences described in the test pattern, it is possible to generate unwanted incidents representing these consequences. By dragging the unwanted incidents to output ports of *threat interface* instances it is possible to map the unwanted incidents to components of the system that is analyzed.

In order to select the most critical scenarios to be tested in the first place, our tool can calculate likelihood values for dependent incidents using Monte Carlo Simulation. Initially at least CWE derived vulnerabilities contain some initial default likelihood value that can be used for such calculations.

The next steps in our process are test case generation and test execution. Actually the so far modelled graph with attached test patterns might already contain sufficient information to create and execute test cases automatically. If additional information is required, e.g. to clarify the mapping to the input ports of system under test if different mappings are possible, then our tool will ask the user to make some manual input.

How exactly the test generation and execution work will be shown in more detail below. The next step in our process is to update the risk graph with information obtained from the security testing. Metrics are required to calculate likelihood values based on the occurrence of the associated unwanted incidents observed in the tested system. The metrics should contain functions or tables setting test results and test coverage in relation to probability values per usage value (e.g. analyzed time span). Ideally test patterns contain sound metrics. Given such metrics, our tool calculates likelihood values based on the tests and updates the risk graph with these values if the user decides to do so.

In addition to the associated unwanted incidents that have already been identified in the risk analysis and that are explicitly monitored in the security testing process, unexpected things might happen as a result of security testing, too. For example, an unexpected kind of exception could be thrown by the system under test. If our tool monitors unexpected behavior, the tool generates new unwanted incidents expressing the previously not expected behavior, which can then be inserted in the risk graph by drag and drop. The relation from the threat scenario which was tested is automatically added.

5.1 Example: Identifying and Testing Risk of Integer Overflows

In order to explore the potential as well as the limitations of our tool and to improve it further, we have created multiple sample program libraries and applications just to analyze and test them.

By dragging the *threat interface* icon to the risk graph drawing area, our tool allows to load system information from compiled programs, libraries or source code. Currently, .Net binaries and sources are fully supported. In the future, it will also analyze components from COM libraries and Java sources. For demonstration, we choose a C# written library. Our tool uses reflection to get information about exported types, functions and to generate partial *threat interfaces* for the elements the user chooses.

In our example we choose to analyze a static function from our sample library called PrintNextNumberToString. For its one and only input parameter of type ‘signed 32 bit integer', in the menu of that input port control, our tool automatically suggests a vulnerability "Integer Overflow or Wraparound", which was generated from CWE-190. The vulnerability contains an initial likelihood value "Medium" because CWE-190 says this is the likelihood that such a weakness is exploited. This likelihood information can be used for identifying the priority of testing related threat scenarios.

The menu of the CWE based vulnerability "Integer Overflow or Wraparound" contains suggestions for potentially related threat scenarios that correspond to CAPEC attack patterns. In the example, the threat scenario "Forced Integer Overflow" is suggested, which is based on CAPEC-92. The analyst can insert the threat scenario by dragging it to the risk graph. The relation from the vulnerability "Integer Overflow or Wraparound" to the "Forced Integer Overflow" threat scenario is automatically added.

The menu of the threat scenario contains a list of all applicable test patterns from our test pattern library. Each test pattern contains a list of unwanted incidents that might be the result of executing an instance of the test pattern. For test patterns related to "Forced Integer Overflow" there is an unwanted incident called "Unhandled arithmetic overflow". The unwanted incident can be dragged to the risk graph. Typically it will be added to some output port of a function in a *threat interface* instance where the unwanted incident could be detected.

Currently there are two test patterns available in the library of our tool to test for forced integer overflows. They differ in their strategies, directives and metrics. One test pattern only generates the extreme and special integer values like maximum, minimum and zero. The second test pattern additionally uses a data fuzzing strategy and creates a certain number of random test values. The generator of the second test pattern has multiple optional parameters. One can be used to directly set the number of fuzz test cases that should be generated. Our tool allows setting this parameter manually. However there are also parameters available that forward values from the risk analysis and then the test pattern calculates the number of test cases that should be generated based on these values. These parameters are namely values for estimated likelihood, uncertainty, impact on the entire risk picture and the potential consequences. The higher these values are, the more test cases are generated. All values are optional. Hence, it is for example no problem if no consequence values have been estimated so far.

Both test patterns use the same idea to test for integer overflows: First, two different versions of the component that should be tested are generated. One *test version* that will throw an exception on any arithmetic integer overflow unless the code explicitly prevents it and one unmodified *release version*. If the source code is available, this is easy for any .Net program: Arithmetic overflow exceptions can be activated by a compiler switch. The tool generates the *test version* without requiring manual actions. Note: Though not yet implemented, it would principally also be possible to generate a *test version* from an IL assembly, so it could be done without access to source code, too.

Each test value is first tried with the *test version* of the component that will throw arithmetic overflow exceptions by default. If an overflow exception is thrown, then the same test value is tested against the *release version* that does not throw arithmetic overflow exceptions by default. If again the overflow exception is observed, then the *release version* detects the overflow correctly. Of course, when the tested function is called, the overflow exceptions must be treated properly, but throwing the exception itself in the *release version* is not considered to be an error, it is not necessarily an unwanted incident. If treated correctly by the caller, the program might continue without problems.

In contrast if only the *test version* throws an overflow exception and the *release version* does not throw an overflow exception, then the *release version* calculates eventually wrong values and there is probably no way to detect the error.

Besides detecting arithmetic overflow exceptions, any other exception that occurs during the test is regarded to be an unexpected exception and it is reported as a potentially new unwanted incident to our tool. Fig. 4 shows how the testing is evaluated.

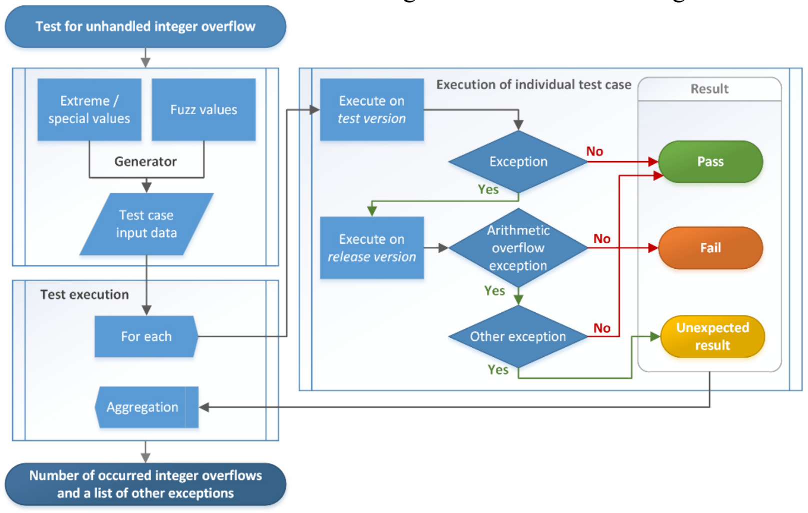

Test for unhandled integer overflow
Extreme / special values
Fuzz values
Generator
Test case input data
Test execution
For each
Aggregation
Number of occurred integer overflows and a list of other exceptions
Execution of individual test case
Execute on test version
Exception
Execute on release version
Arithmetic overflow exception
Other exception
Yes
No
Result
Pass
Fail
Unexpected result

Fig. 4. Testing for Integer overflows

Our tool compiles code taken or automatically generated from test patterns. For actually executing the test cases, the risk graph is evaluated to identify which functions have to be called with which parameters and what has to be monitored. In the example we present here, this requires no additional manual work at all.

In our sample library there are multiple functions that can be tested for forced integer overflows. Some are more complex and do expect more than just a single input value. To test such functions, it is necessary to assign test case generators to each input parameter. This can be done by assigning threat scenarios and by choosing test patterns. Then only one of the test patterns is actually tested at a time, i.e. its test strategies are compiled and executed. From the other test patterns, only the test data generators are used. Since by default all possible combinations of the generated test values for each parameter are tested, the amount of test cases can grow very quick by doing so. Alternatively, our tool also allows to create data generators that just produce test data. However, these do require manual writing of code. Finally, there are constant values assignable in an easy way.

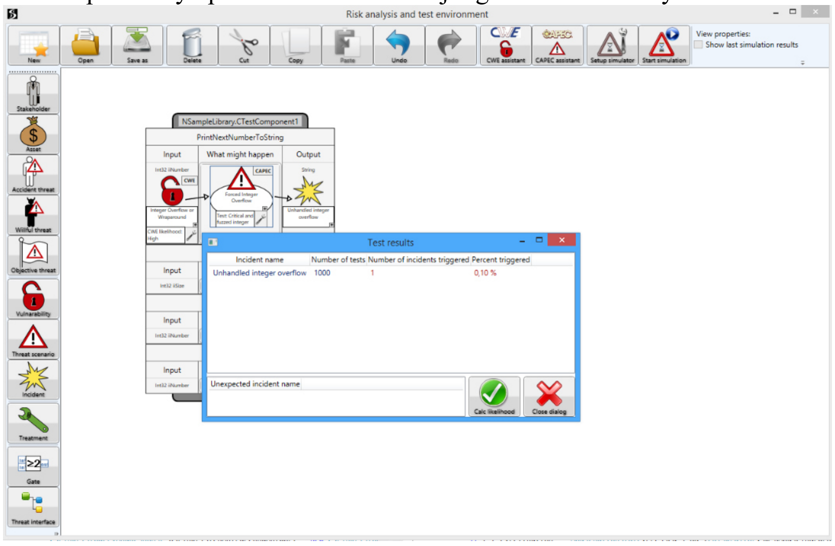
Both test patterns from our library that are applicable to test for integer overflows contain a metric which can be used to calculate the likelihood that an attacker will produce an integer overflow by calling the tested function. For the test pattern that uses fuzzing, the metric uses the total number of test cases that are generated and executed to calculate coverage. The risk graph can be updated with the new likelihood value if the analyst confirms it.

5.2 First Evaluation

In the optimal case, our tool assists the risk analysts and testers so far that only a few mouse actions are required to do combined risk analysis with automated security tests. The example shows that no single line of code has to be written, no test cases have to be created manually and no manual interpretation of the test results is required if a well-defined test patterns are available and applied correctly.

Using our tool, modeling of the risk graph remains a manual task. But the analyst is great much assisted with existing artifacts from libraries and artifacts generated based on test results. Likelihoods can automatically be calculated based on test results. Furthermore, for dependent incidents, likelihood values can be calculated.

Though our tool is in an early phase of development, it already provides an intuitive workflow. Fig. 5 shows the UI of our tool. It proves to save some amount of work and to reduce dependency upon the skills and the judgment of the analysts and testers.



Incident name	Number of tests	Number of incidents triggered	Percent triggered
Unhandled integer overflow	1000	1	0,10 %

Fig. 5. Screenshot of the tool analyzing and testing the example

Currently our test pattern library is pretty small and it is quite a challenge to develop test patterns that are general and flexible enough to be applicable for diverse systems and that can though be instantiated without lots of manual work – especially without manually writing code. Hence, practical usability is for now limited to relatively few cases.

The tool we developed is a standalone application. However its core is an API which can be integrated in and used by other tools. Indeed it will be possible to use our tool only for parts of the combined risk analysis and security testing process.

6 Conclusion, Ongoing and Future Work

Though there are lots of technologies and tools for risk assessment and security testing, applying them for large complex systems is still a challenge.

Development of the methodology and the tool described here is still in an early stage. Our efforts are driven by case studies. These provide use cases and requirements inspiring our development. We plan to test and to evaluate our method and our tool by using them within these case studies. Additionally we will analyze the same use cases with other existing methods and tools so that we can compare the results in relation to the effort for the different approaches.

Our vision is that risk assessment should become a process that typically takes place in an open collaboration. Risk analysis results and test patterns should be made accessible for anybody as reusable artifacts. We plan to create a public open database for that purpose. This data would be helpful for other developers reusing the analyzed component as they could integrate the risk analysis artefacts in their own compositional risk assessment for their products. Reusing the test patterns could reduce testing costs and improve testing quality. The end users could benefit from such a database, too, because they could inform themselves about the remaining risks in a standardized way.

7 References

1. International Organization for Standardization: ISO 31000 Risk management – Principles and guidelines (2009)
2. International Organization for Standardization: ISO Guide 73 Risk management – Vocabulary (2009)
3. Bouti, A., Kadi, D.A.: A state-of-the-art review of FMEA/FMECA. International Journal of Reliability, Quality and Safety Engineering 1, 515–543 (1994)
4. International Electrotechnical Commission: IEC 61025 Fault Tree Analysis (FTA) (1990)
5. International Electrotechnical Commission: IEC 60300-3-9 Dependability management – Part 3: Application guide – Section 9: Risk analysis of technological systems – Event Tree Analysis (ETA) (1995)
6. Lund, M. S., Solhaug, B., Stølen, K.: Model-Driven Risk Analysis – The CORAS Approach. Springer (2011)
7. Lund, M. S., Solhaug, B., Stølen, K.: Evolution in relation to risk and trust management. Computer 43(5), pp. 49–55, IEEE (2010)
8. Kaiser, B., Liggesmeyer, P., and Mäckel, O.: A new component concept for fault trees. In: 8th Australian workshop on Safety critical systems and software (SCS'03), pp. 37–46. Australian Computer Society (2003)
9. Papadoupoulos, Y., McDermid, J., Sasse, R., and Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Reliability Engineering and System Safety, 71(3), pp. 229–247, Elsevier (2001)
10. Erdogan, G., Li, Y., Runde, R. K., Seehusen, F., Stølen, K.: Conceptual Framework for the DIAMONDS Project, Oslo May (2012)
11. Erdogan, G., Seehusen, F., Stølen, K., Aagedal, J.: Assessing the usefulness of testing for validating the correctness of security risk models based on an industrial case study. Proc. Workshop on Quantitative Aspects in Security Assurance (QASA'12), Pisa (2012)

12. Benet, A. F.: A risk driven approach to testing medical device software. In: Advances in Systems Safety, pp. 157–168. Springer (2011)
13. Kloos, J., Hussain, T., and Eschbach, R.: Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In: Software Testing, Verication and Validation Workshops (ICSTW 2011), pp. 26–33. IEEE (2011)
14. Viehmann, J.: Reusing Risk Analysis Results - An Extension for the CORAS Risk Analysis Method, 4th IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT 2012), Amsterdam 2012, pp. 742-751, IEEE 2012
15. G. J. Bach, "Heuristic Risk-Based Testing", Software Testing and Quality Engineering Magazine, November 1999, pp. 96-98.
16. H. Stallbaum and A. Metzger, "Employing Requirements Metrics for Automating Early Risk Assessment", In: Proceedings of the Workshop on Measuring Requirements for Project and Product Success, MeReP07, at Intl. Conference on Software Process and Product Measurement, Spain, 2007, pp. 1-12.
17. H. Stallbaum, A. Metzger, and K. Pohl, "An Automated Technique for Risk-based Test Case Generation and Prioritization", In: Proceedings of 3. Workshop on Automation of Software Test, AST'08, Germany, 2008, pp. 67-70.
18. T. Bauer et al., "From Requirements to Statistical Testing of Embedded Systems", In: Software Engineering for Automotive Systems, (ICSE), 2007, pp. 3-10.
19. F. Zimmermann, R. Eschbach, J. Kloos, and T. Bauer, "Risk-based Statistical Testing: A Refinement-based Approach to the Reliability Analysis of Safety-Critical Systems", In: Proceedings of the 12th European Workshop on Dependable Computing (EWDC), France, 2009.
20. Y , Chen,. R. Probert, and P. Sims, "Specification-based Regression Test Selection with Risk Analysis", In: Proceedings of the 2002 conference of the Centre for Advanced Studies on Collaborative research (CASCON '02), 2002, pp. 1.
21. Object Management Group (OMG): UML Testing Profile. URL: http://www.omg.org/spec/UTP
22. Utting, M.; Legeard, B.: Practical Model-based testing – A Tools Approach, Elsevier, 2007.
23. Smith, B.: Security Test Patterns (2008). http://www.securitytestpatterns.org/doku.php
24. Feudjio, A. V.: Initial security test patterns catalogue. DIAMONDS project deliverable D3.WP4.T1
25. MITRE: Common Attack Pattern Enumeration and Classification (2014). http://capec.mitre.org
26. MITRE: Common Weakness Enumeration (2014). http://cwe.mitre.org
27. International Organization for Standardization/: ISO/IEC 29119-1 Systems and software engineering—Software testing—Part 1: Concepts and definitions (2013)