Towards a Systematic Identification of Security Tests Based on Security Risk Analysis

Jan Stijohann[1] and Jorge Cuellar[1] (Supervisor)

Siemens AG, Germany,
firstname.lastname@siemens.com

Abstract. Today's security testing is not systematic much less standardized. In particular, there are no clearly defined criteria for selecting relevant tests. Thus different analysts come to different results and sound quality assurance is hardly possible. Literature suggests basing the choice and prioritization of tests on risk considerations but lacks a systematic approach for a traceable transition from abstract and business-oriented risk analysis into the concrete and technical security testing world. We aim at bridging this gap in two steps: The first one bridges between high-level and non-technical "business worst case scenarios" and less abstract "technical threat scenarios" using a technical description of the system and a systematic STRIDE-based elicitation approach. The second is a rule-based step that maps technical thread scenario to "test types", that is, to classes of tests that need to be adapted to the particular system under validation. Our method provides traceability for the choice for security tests and a standardized minimum quality assurance level.

Keywords: Security, risk driven testing, risk analysis, threat modeling

1 Introduction

Today's security testing is not a systematic or standardized process: a tester has neither clearly defined criteria for choosing or prioritizing possible tests, nor when to stop. The results of such security testing sessions depend on the tester's know-how, experience, intuition, and luck. Thus different people come to different results and a sound quality assurance is hardly possible.

Several standards and scientific papers suggest to base the choice and prioritization of tests on risk considerations. Yet only few explain how this can be done on a *technical* level, e.g. [8], and there is a lack of concrete guidelines on how to systematically select security tests based on *business*-oriented risk analysis [3].

However, if the risk analysis remains at a high level, it misses essential risk factors and is of little help for a subsequent security testing. If, on the other hand, it includes technical threats, such as "Buffer Overflow" or "SQL Injection", those choices seem arbitrarily taken, that is, independently of the unique, specific characteristics of the system under verification (SUV). It is not clear why they concern this particular SUV, nor why other similar threats with the same possible impact are ignored. For instance, in case of the buffer overflow, an

analyst may simply not consider "format string vulnerabilities", "double frees" or "null pointers", just because he is not aware of their existence, while "buffer overflow" is a common vulnerability that he knows.

Objectives This paper aims at bridging this gap between risk analysis and security testing, offering a systematic approach for a traceable transition from abstract, business-oriented risk analysis to the concrete and technical security testing world. The process must start from concrete technical information about the SUV and result in a set of tests that may still require some adaptation, but are manageable by experienced security testing experts. Moreover, it should be applicable in real-world industrial environments and provide a clear benefit for the security analyst, in terms of effort assurance, and transparency.

Outline of our Solution We propose a two-step-process for a systematic identification of security tests based on risk analysis:

1. The first one goes from high-level and non-technical "business worst case scenarios" to less abstract "technical threat scenarios" via a systematic *STRIDE*[1]-based elicitation approach. This step requires a sufficiently technical security overview, in the form of a Data Flow Diagram (DFD) of the SUV, annotated with security relevant information.
2. The second step derives security tests from the technical threat scenarios. It guides the analyst with rules that map patterns in the DFD to "test types", which are test templates which need to be instantiated, that is, adapted to the implementation, configuration and state of the SUV. In addition to those re-usable mapping rules, the selection of appropriate tests is supported by organizing the test types in a "test library" and tagging the entries according to the tested system element, the tested security property, the technology, and the sophistication of the test.

2 Current Work

2.1 Technical System Description

The technical system description captures and structures the security relevant technical aspects of the SUV in a comprehensive and systematic way. The resulting *security overview* is crucial for the transition from risk analysis results to security tests (see Section 2.2 and 2.3) and it provides the technical system information needed to identify and instantiate appropriate test types. The security overview should have the following properties:

Created by the Security Analysis Team. Design documents are often not suitable as a security overview as they tend to be overwhelming, out-dated, incorrect, incomplete, or they miss relevant security information. It is therefore judicious to let the risk analysis team create its own suitable, that includes

[1] The acronym is introduced as part of Microsoft's STRIDE threat modelling [7]; it stands for spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege.

sufficiently technical, security overview. Besides studying existing documents, the security analysts should consider interviews and white-board sessions with developers, as well as tool-supported approaches (as explained later on).

Sufficiently Technical. Examples of security relevant technical information that a security overview should consider in detail are: Data flow related aspects (including interfaces and trust boundaries), security controls (such as authentication, encryption, or input validation), sensitive information that must be protected, relevant design properties (protocols, sanitization/encodings, file permissions process privileges, etc.).

Generated with Tool Support. Manually creating a correct and sufficiently technical security overview can become time consuming and tedious. This is especially the case for complex and dynamically evolving software where no one has the complete overview. Reconnaissance tools, such as port scanners, network sniffers or static and dynamic analysis tools, can help to obtain and keep the overview. As a side effect, the tool-based generated results can be used to discover discrepancies with existing design documents and interview results.

Presented in a Syntactically Standardized Language. One possible standardized graphical language is given by Data Flow Diagrams (DFD) as used in [7], annotated with additional security-relevant information. DFDs are well suited for security analysis as they contain the interfaces that an attacker may use and describe how data, often the target of attack, moves through the system.

2.2 STRIDE-based Elicitation of Technical Threat Scenarios

A STRIDE-based elicitation of technical threat scenarios is the first step for the transition from high-level risk analysis into the practical security testing world. Starting point are short, informal, and non-technical descriptions of *business worst case scenarios* (BWCSs). Most risk analysis methods include the definition of BWCS or similar equivalents.

The security analyst examines which security violations of system elements could lead to the BWCSs. We call the tuple (*system element, violated security property*[2]) a *technical threat scenario* (TTS). The mappings of BWCSs to TTSs are created top-down (given a BWCS, examine which combination of TTSs could lead to it) and bottom-up (for each DFD element, check if a violation of any security property, in combination with other TTSs, could lead to a BWCS).

2.3 Mapping Technical Threat Scenarios to Test Types

The mapping of BWCSs to TTSs is only the first step towards security tests. The TTSs are still too abstract and need to be further concretized. For this purpose, we suggest the concept of *test types*. A test type is a template for a class of tests which a security analyst can instantiate into an executable test by adapting and completing it according to the implementation, configuration and state of the SUV.

[2] represented by one letter from the STRIDE acronym

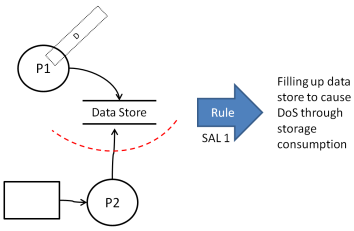
DFD-Pattern-based Rules One way to capture and leverage the security testing expertise required to derive appropriate test types from TTSs is via mapping rules. We suggest such rules to consist of the following elements:

- A pattern in an annotated DFD. Besides a mandatory TTS which includes the security property violation, the pattern can include additional system elements and further annotations.
- The level of sophistication for the security tests. It is determined by risk considerations such as the expected attacker and the desired assurance.
- A reference to the suggested test type that fits to the above characteristics.

Rules of this structure allow the test type derivation to become systematic and traceable. The mapping rules suggest an initial set of test types which helps to achieve a minimum quality standard.

Our DFD-patterns-based rules are inspired by EMC's "Threat library" [2], but in contrast our method is a) to be used during testing not development, b) uses explicit rules, c) considers SALs and security violations, and d) yields concrete test types instead of abstract threats.

Test Library The presented concept of mapping rules anticipated the idea of a re-usable collection of test types. We suggest that the entries of such a *test library* consist of a title, a textual description, and the information needed to be matched by mapping rules. The latter allows to filter the library entries according to the targeted system element, the security property to violate, the technology, and the sophistication of the test. This supports security analysts that want to go beyond the mere rule-generated minimum set of test types. Figure 1 shows an exemplary mapping rule and an excerpt of the current test library.



(a)

Name	SAL	Example Tools	Sys. Elem.	STRIDE
Reverse Engineering			**Process**	**I, E**
Pattern matching in hex editor	1	HxD	Process	I, E
Static binary analysis with disassembler	3	objdump	Process	I, E
Analysis of decompiled high-level code	2	.NET reflector	Process	I, E
Analysis of intermediate code	3		Process	I, E
Analysis with 3rd party tools	2	Sysinternals, strace, strings	Process	I, E
Dynamic binary analysis with debugger	2	OllyDbg	Process	I, E
Dynamic binary analysis with custom debugging scripts	3	IDA Pro scripts, pydbg	Process	I, E
Fuzzing	-		**Process**	**D**
Fuzzing with "point and shoot" tools	1	VolPer, 4n FTP Fuzzer	Process	D
Dumb fuzzing	1	/dev/random, netcat	Process	D

(b)

Fig. 1. (a) A rule maps a DFD pattern to a test type. (b) Excerpt of the test library.

2.4 Evaluation of Current Results

The steps of our method can be integrated in an ISO 31000 conform risk analysis process, provided that it estimates risk based on clearly defined threat scenarios. The suggested security overview is obtained in the **context establishment**

phase, together with the identification of non-technical assets and their security requirements. Based on this, the BWCSs are derived in the **risk analysis** phase and are then mapped to technical threat scenarios. The determination of expected threat agents including their estimated skill level is also part of the risk analysis phase. During the **risk estimation** phase, the likelihood of the technical threat scenarios is estimated and risk values for the BWCSs are assigned. The subsequent **risk evaluation** allows to prioritize the TTSs, which are then, as part of the **risk treatment**, covered by appropriate tests.

3 Future Work

3.1 Extending the Collection of Mapping Rules

We want to extend our current rule set in order to cover different SUVs from different application domains. For this purpose, we intent to proceed according to the following three-step procedure:

1. Identify classes of vulnerabilities to be covered. Appropriate sources are lists of threats and vulnerabilities such as CAPEC and CWE, and secure software development and security testing literature, such as [6], [4], and [5].
2. For each vulnerability class, analyse the technical context and identify suitable environment properties, in order to determine a reliable pattern that indicates the possible presence of the vulnerability.
3. Determine how the identified context information can be obtained, how it can be represented in form of a DFD pattern, and which (possibly new) test types match these patterns.

3.2 Tool Support

Our goal is to further automate the technical system description. We therefore plan to evaluate more advanced tools such as scriptable debuggers (e.g. pydbg, IDA Pro, Immunity Debugger), advanced dynamic analysis tools (e.g. WinApiOverride32, Microsoft Application Verifier), or operating system utilities (e.g. strace, eventlog). First steps are described in [8].

Once we have extended our rule base, a manual application of rules for the derivation of test types may become tiresome and inefficient. Our idea is to develop a tool that, similar as described in [1], processes annotated DFDs, detects the patterns and finally outputs adequate test types.

3.3 Further Evaluation

We plan to apply the presented method, together with a light-weight ISO 31000 conform risk analysis, in future real-world security assessments. The idea is to develop a questionnaire to capture observations after each assessment and thus support a systematic evaluation regarding benefits, drawbacks, practicability, areas for improvement, and inherent limitations.

We intend to analyse to what extend the tests, which have been identified using our method, satisfy the following requirements: 1) the security test addresses at least one BWCS. 2) it targets the proper system element and aims at violating the right security property, 3) it has the proper level of sophistication with respect to the expected threat agents 4) it reflects the technology, implementation and configuration of the SUV, and 5) the test priority is high enough with respect to the given time and budget.

4 Conclusions

This work proposes a tool-supported method to bridge the gap between a risk analysis and a corresponding security testing.

The presented method requires a security analysts to critically accompany the involved steps and adapt, possibly manually complement, and interpret the intermediate results. However, our method can guarantee a certain quality standard and, first and foremost, will make security testing more traceable for all involved parties including security testers, developers, and managers.

5 Acknowledgments

This work was partially supported by the EU Project SPaCIoS (FP7 257876, Secure Provision and Consumption in the Internet of Services) and NESSoS (FP7 256890, Network of Excellence on Engineering Secure Future Internet Software).

References

1. M. Abi-Antoun, D. Wang, and P. Torr. Checking threat modeling data flow diagrams for implementation conformance and security. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 393–396. ACM, 2007.
2. D. Dhillon. Developer-driven threat modeling: Lessons learned in the trenches. *IEEE Security and Privacy*, 9(4):41–47, July 2011.
3. DIAMONDS Consortium; F. Seehusen (Editor). Initial methodologies for model-based security testing and risk-basedsecurity testing. http://www.itea2-diamonds.org/Publications/2012/index.html, Aug. 2012. Project Deliverable D3.WP4.T2_T3.
4. M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 11 2006.
5. T. Gallagher, L. Landauer, and B. Jeffries. *Hunting Security Bugs*. Microsoft Press, 6 2006.
6. M. Howard, D. LeBlanc, and J. Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill Osborne Media, 9 2009.
7. M. Howard and S. Lipner. *Security Development Lifecycle*. Microsoft Press, 11 2009.
8. C. Wysopal, L. Nelson, D. D. Zovi, and E. Dustin. *The Art of Software Security Testing: Identifying Software Security Flaws*. Addison-Wesley Professional, 11 2006.