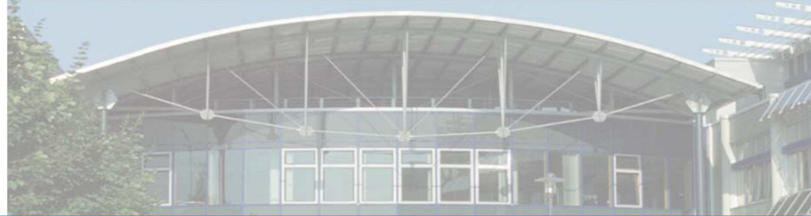
softScheck
we identify vulnerabilities others don't



Software Security

Cost-Effective Identification of Zero-Day Vulnerabilities with the Aid of Threat Modeling and Fuzzing

White Paper 22. June 2011

Prof. Dr. Hartmut Pohl
Geschäftsführender Gesellschafter softScheck GmbH Köln
Hartmut.Pohl@softScheck.com
www.softScheck.com

Cost-Effective Identification of Zero-Day Vulnerabilities with the Aid of Threat Modeling and Fuzzing

Content

Content 1

1. Introduction to Threat Modeling and Fuzzing...1

2. There Is No Such Thing as Bug-Free Software..2

3. Software Development Life Cycle 2

4. Threat Modeling 3

5. Static Analysis..4

6. Dynamic Analysis 5

7. Conclusion..7

Further Reading..8

1. Introduction to Threat Modeling and Fuzzing

The use of such tools as Threat Modeling (in the design phase) and Fuzzing (in the verification phase) enables the cost-effective identification of hitherto undisclosed vulnerabilities. This is because the effort required to fix and patch vulnerabilities is considerably lower in the early stages of software development (requirements and design phase) than in the release phase, when the software has already been delivered to the customer (cf. Fig. 1 - Lifecycle of Secure Software Development). Furthermore, these methods can be used in all areas of application: individual software, standard software such as ERP, CRM software and company-specific extensions, operating systems, web browsers, web applications, network protocols, etc.

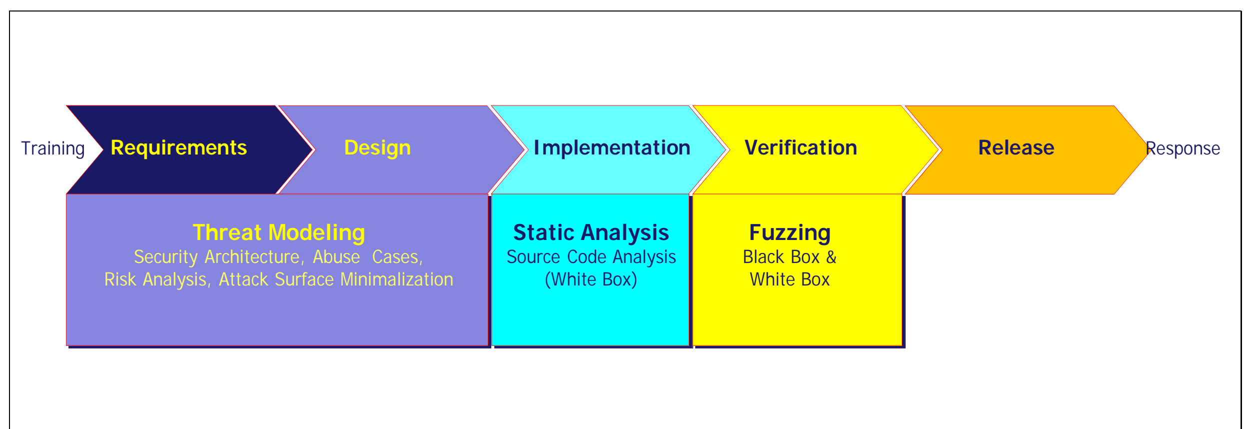


Fig. 1: Lifecycle of Secure Software Development

Conventional methods of identifying and fixing vulnerabilities – especially undisclosed (less-than-zero-day) vulnerabilities – are very expensive; moreover, many vulnerabilities are detected only after the software has been delivered to the customer – sometimes by third parties.

Fig. 2 ('Cost of Fixing Vulnerabilities in the Software Development Lifecycle') illustrates the exponentially rising cost of fixing (and patching) vulnerabilities during the individual stages of development before release.

In fact, several studies, such as the one conducted by the National Institute of Standards and Technology (NIST), claim that the cost of patching increases a hundredfold from the design phase to the release phase. Our own studies show that the use of Threat Modeling and Fuzzing is a very cost-effective way of identifying vulnerabilities, even after the software has been released to the market.

Vulnerabilities can be detected as follows:

- Systematic (tool-based) identification of vulnerabilities with the aid of Threat Modeling and Fuzzing
- Tool-based identification of the most essential and severe vulnerabilities, i.e. those that can easily be exploited from a remote server over the Internet, as well as evaluation (prioritization) of the remaining vulnerabilities.

Threat Modeling requires design documentation. Fuzzing does not require a source code to achieve good results: The software is examined while it is being executed (Black Box Testing). To this end, the software can be executed on a virtual machine or another testing system.

2. There Is No Such Thing as Bug-Free Software

It is impossible to develop bug-free software. This makes it necessary to conduct software reviews, which means there is a close link to quality assurance. Due to the complexity of the code (e.g. the number of lines it includes), it is mostly impractical to carry out manual checks. Tests are exclusively aimed at examining specified functionalities. Non-functional tests are neglected.

With the aid of Threat Modeling and Fuzzing, it is possible to identify all the vulnerabilities that support the following attacks by exploits:

- Breach of access rules
- Format string attacks
- SQL injections
- Buffer overflows
- ...

To achieve this, Fuzzing deliberately aims at providing unspecified, random data to the attack surface, which causes the software to malfunction. Threat Modeling also reveals design errors, for instance, by using and evaluating attack trees and data flow diagrams.

3. Software Development Life Cycle

The techniques that are used to develop software nowadays are unable to avoid vulnerabilities completely - human errors cannot be ruled out; even though programming guidelines may be in place, they are not (strictly) adhered to and (entirely) audited. And yet, there are effective tools - operating far beyond the scope of classic testing technologies – that detect vulnerabilities as early as during the design phase or at the latest during the verification phase (Fuzzing and Penetration Testing). Still, many vulnerabilities are detected only after the software has been delivered to the customer (release). The following will illustrate the methods used to identify vulnerabilities, as well as mentioning some well-known tools.

The cost of eliminating software bugs depends on the time of their detection within the Software Development Life Cycle (SDLC). If the bug is only identified after release (i.e. by the customer), the cost increases by a factor of one hundred (cf. Fig. 2: Cost of Fixing Vulnerabilities in the Software Development Lifecycle). The quality of software products is often determined by a shortage of resources in the company where the products have been developed. In addition, the market calls for very short software life cycles.

Hence, the use of such methods as Threat Modeling and Fuzzing to identify vulnerabilities meets market requirements. Compared to traditional testing techniques, these methods do not require many resources – as has been confirmed by research conducted during softScheck projects.

Fuzzing and Threat Modeling enable software developers, users and customizers to enhance the efficiency and cost-effectiveness of software tests by using appropriate tools. Thus they also improve software security: the tools help reveal hitherto unknown bugs and vulnerabilities.

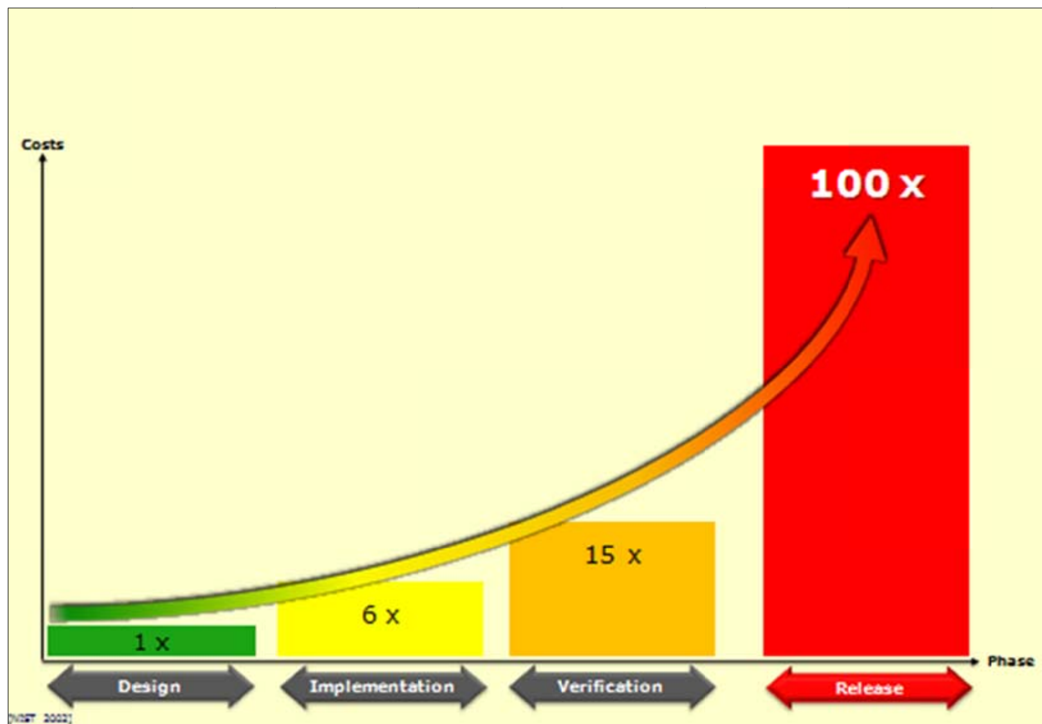


Fig 2: Cost of Fixing Vulnerabilities in the Software Development Lifecycle

softScheck analyzed and continously evaluates more than 300 tools for Threat Modeling and Fuzzing available worldwide Our work is based on a successfully completed research project that was funded by the German Federal Ministry for Education and Research and carried out at the Bonn-Rhine-Sieg University of Applied Sciences (code: FKZ 01 IS 090 30).

4. Threat Modeling

This proactive and heuristic method advances the methodological development of a trustworthy system design or architecture during the design phase, which makes it the most cost-effective way of fixing vulnerabilities.

At the same time the existing system design and architecture can be verified – with the aim of identifying, evaluating and fixing vulnerabilities.

Further goals include understanding the security architecture, identifying design bugs and minimizing the number of possible attack surfaces.

Examples of Threat Modeling Tools:

- Microsoft Threat Analysis & Modeling
- Microsoft SDL Threat Modeling Tool
- Trike

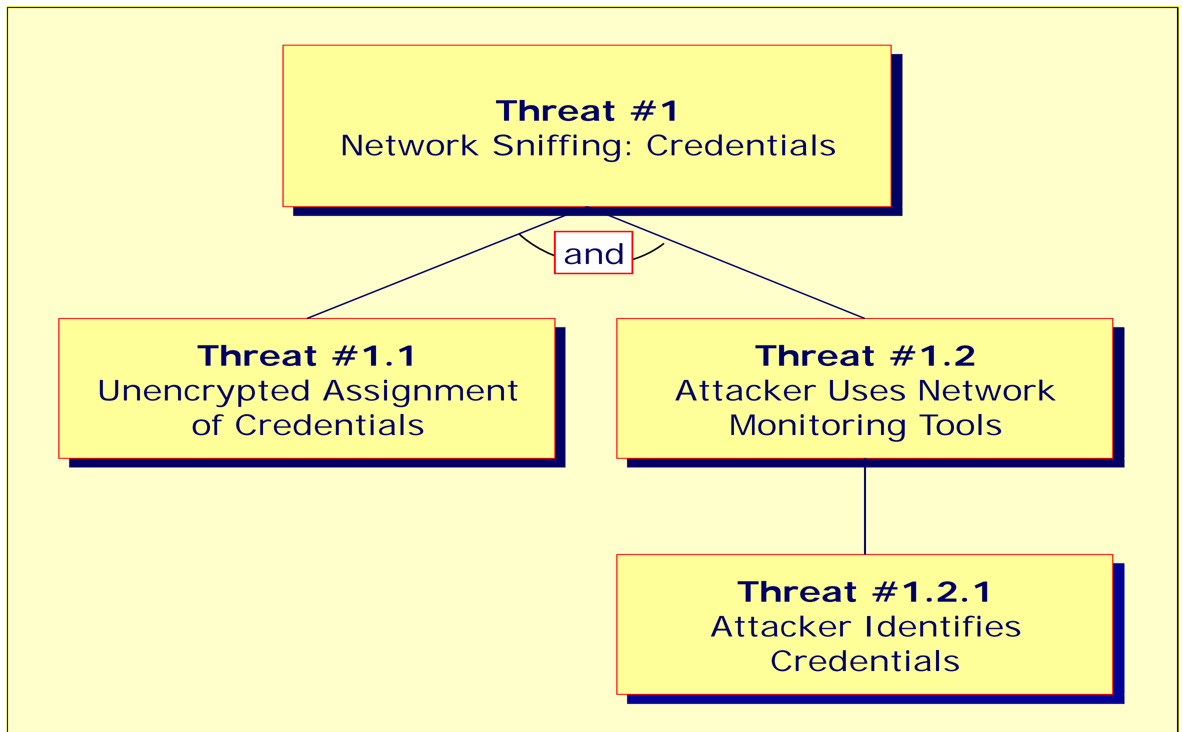


Fig. 3: Threat Modeling - Attack Trees

The number of vulnerabilities identified through Threat Modeling is significant. For instance, in one of the authors' projects 1 critical and 29 high-ranked vulnerabilities (that could have been exploited over the Internet) were identified as early as in the design phase (cf. Fig. 4: Threat Modeling – Vulnerabilities Identified in Standard Software).

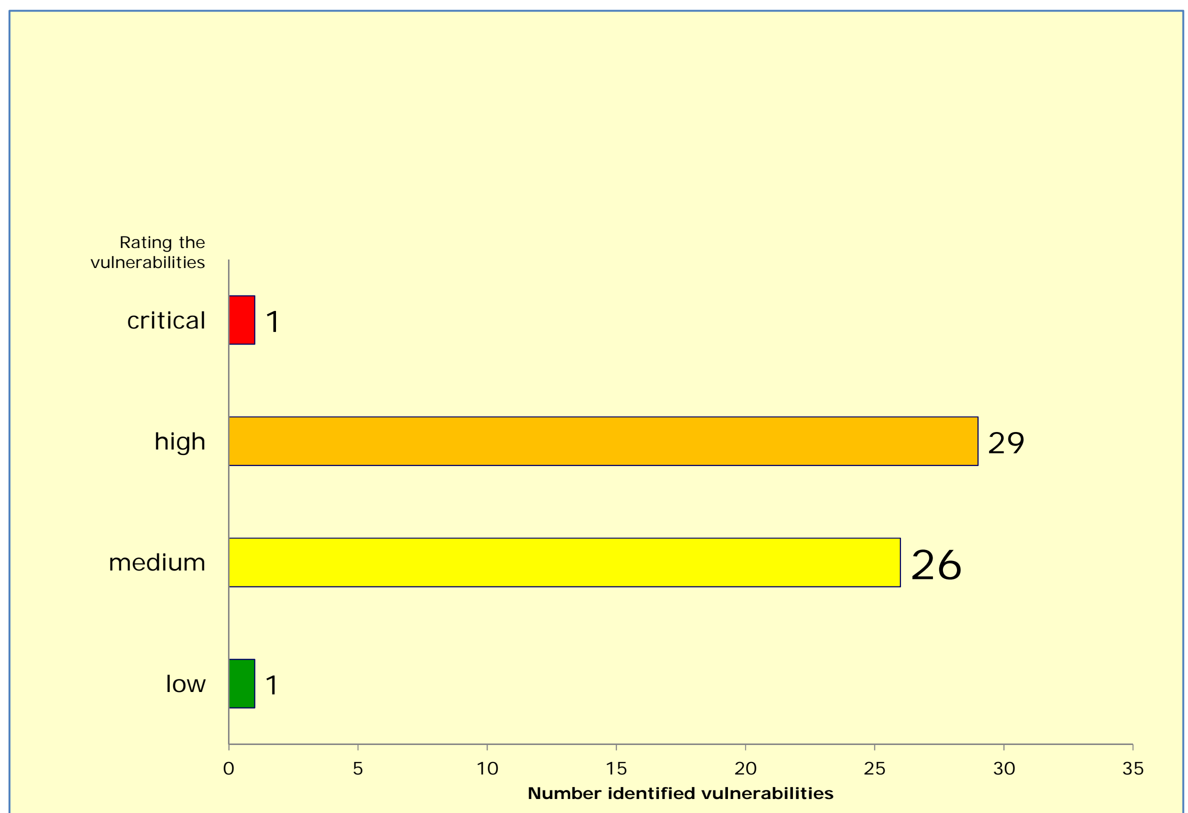


Fig. 4: Threat Modeling – Vulnerabilities Identified in Standard Software

5. Static Analysis

This method serves to analyze the source code of the target software without executing it (in contrast to dynamic analysis techniques, such as Fuzzing). During the implementation phase, the system checks whether the programming language conforms to the programming guidelines – just like a parser that conducts a lexical, syntactic and semantic analysis of the programming code.

Examples of static analysis tools include:

- Klocwork
- Pixy
- XDepend

6. Dynamic Analysis

Fuzzers examine software for its robustness by feeding it with random or targeted data input. In this way, sporadic system failures and unintended data leakage – the most common results of security-related software bugs - can be avoided, which proactively mitigates the harmful consequences of considerable sales shortfalls, data protection problems and reputational damage.

To this end, the Fuzzer identifies the input interfaces to which it sends the input data (fuzz).

The quality of Fuzzing is largely determined by the code coverage of the tested input space (which may be infinite) and the quality of the generated data.

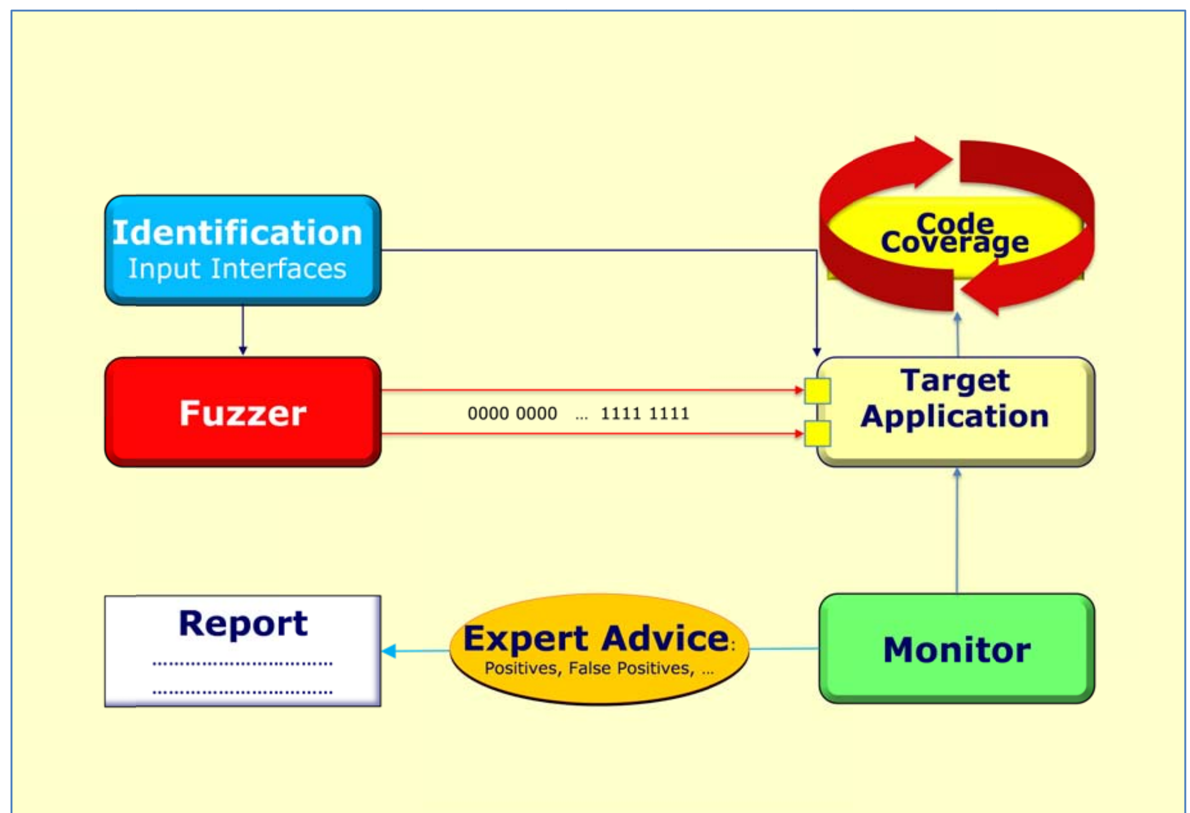


Fig. 5: The Fuzzing Process

The Fuzz Testing Process is part of the SDLC verification phase (cf. Fig. 1: Life Cycle of Secure Software Development). Vulnerabilities should not be fixed later than at this stage because the cost of fixing them increases dramatically after release (cf. Fig. 3: Cost of Fixing Vulnerabilities in the Course of the Software Development Lifecycle).

Once the input interface of the target application has been identified and been provided with the Fuzzer-generated data, a special tool monitors the target application, informing the tester of such anomalies as program failures, high CPU or storage utilization, etc. (cf. Fig. 5 – The Fuzzing Process). After that the system

analyst conducts an analysis to determine the severity of the vulnerability and the extent to which it is reproducible and exploitable over the Internet.

For Fuzzing, it is not necessary to know the source code of the target application, which can facilitate asking the advice of external security analysts. Fuzzers may be used locally or remotely, depending on their type. A Command Line Fuzzer is an example of a local Fuzzer. Network-based applications need to be analyzed by remote Fuzzers. Additionally, there are Fuzzers to test web browser protocols and web applications. Furthermore, Fuzzers may be subdivided into dumb and smart Fuzzers.

Examples of Fuzzing Tools

- AxMan
- beSTORM
- Defensics
- FileFuzz
- FTPStress Fuzzer
- Fuzz
- Peach
- SPIKE
- SPIKEfile

Smart Fuzzers run program-controlled and independent tests of a target application, often without special preparation or support from the user; their use is mostly subject to a license. Often only smart Fuzzers enable the system analyst to penetrate the target application and test the programming code.

Dumb Fuzzers are not able to identify the structure of the target application; instead, they generate unspecified, random input data. As there is no program control, the user needs to have considerable experience of using them. Dumb Fuzzers can often be downloaded from the Internet free of charge (cf. Fig. 6: Divergence Product Costs – Staff Expenditure).

Fuzzing frameworks enable users to develop Fuzzers that are adjusted to their individual needs – similar to a construction kit. However, their cost of development is comparatively high – especially as there are already working Fuzzers available for all common applications. Fuzzing frameworks are suitable for new proprietary target applications - for example, new network protocols. In commercial tools, the command line interface has often been replaced with a graphical user interface.

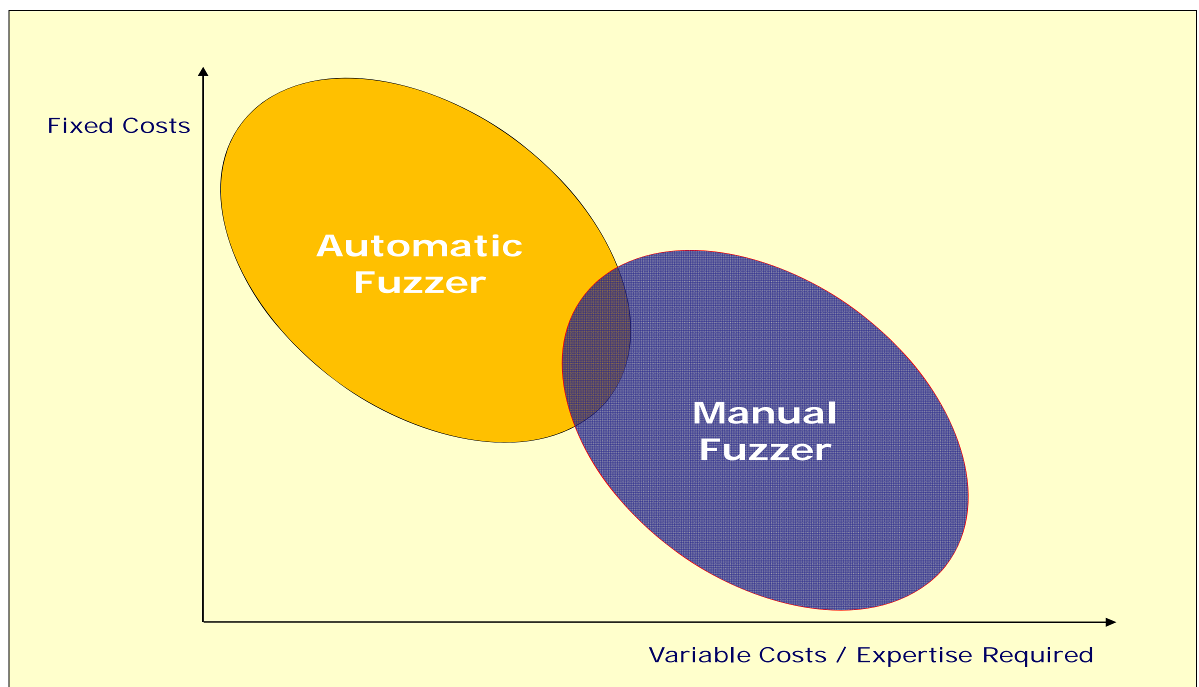


Fig. 6: Divergence Product Costs – Staff Expenditure

7. Conclusion

Fuzzing and Threat Modeling are two different tool-supported methods that can be used to identify software bugs – above all, security-related bugs. For instance, zero-day attacks – which are among the 20 most frequent forms of attacks – can be avoided with the aid of Threat Modeling and Fuzzing. Moreover, these tools help increase the level of application security in an efficient way that meets market requirements. In addition, the individual approach to Fuzzing can be adequately scaled to the ever shorter software development life cycles using the methods available.

Conclusion drawn from employing the methods outlined above

- Systematic search for and successful (!) identification of the essential vulnerabilities
- And that in **each** software application: individual software, standard software such as ERP, CRM and company-specific extensions, operating systems, etc.
- Source Code is not necessary for Threat Modeling and Fuzzing – executable files are sufficient: Black Box Testing

Software developers can achieve a higher return on (security) investment by combining the methods outlined above and integrating them into the Software Development Life Cycle (SDLC). Moreover, they can proactively improve software quality, reduce their times to market and make cost savings. Along with saving costs, they may improve their reputation by developing more secure software.

Threat Modeling and Fuzzing can be used for all applications - from protocols to individual software through to web applications. To support a more demand-actuated choice of suitable tools, softScheck has drawn up a taxonomy that will be published separately.

There is a considerable number of vulnerabilities discovered through Fuzzing (often without any source code). This is the reason for the wider use and popularity of the method.

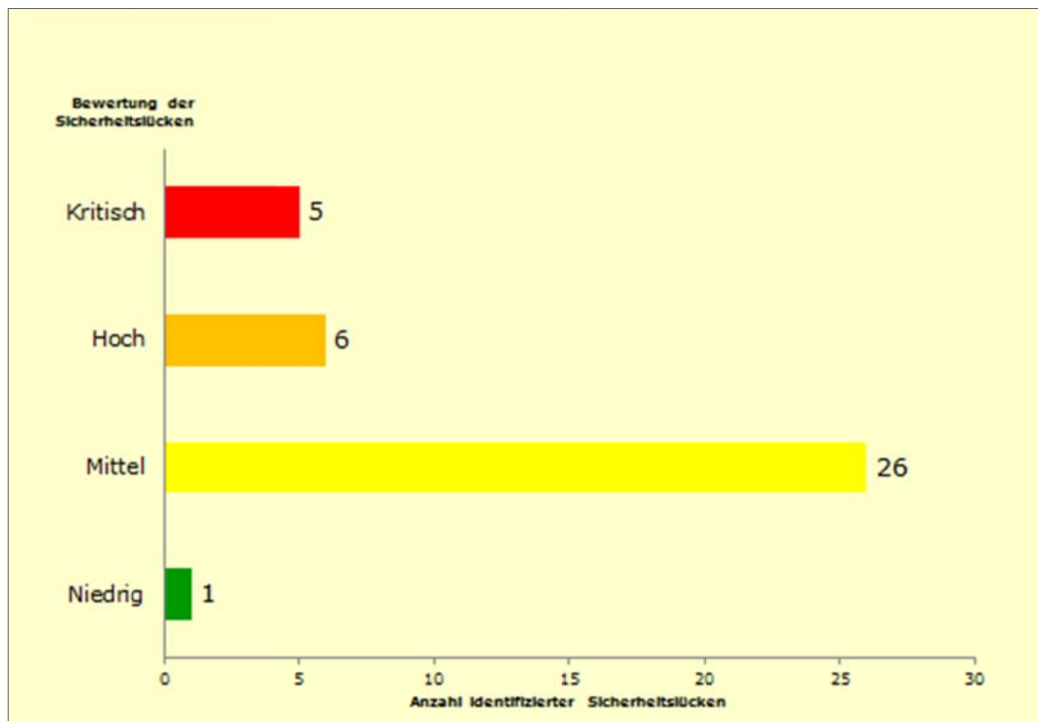


Figure 7: Vulnerabilities Detected through Fuzzing

For example, in another project conducted by the authors, five critical (hitherto undisclosed) vulnerabilities were identified in standard software that had already been delivered to the customer; these vulnerabilities could have been exploited over the Internet (cf. Fig. 7: Vulnerabilities Detected through Fuzzing).

And this is even though the software developer has put in place numerous guidelines that also encompass "secure" software.

The Successful Methods

Threat Modeling

- Vulnerabilities are detected during the design phase.
- Systematic and tool-based methods.
- Vulnerabilities can be prioritized: exploitable over the Internet and/or little effort required by the attacker.

Fuzzing

- Black Box Testing – no source code required – executable files are sufficient!
- Commonly used methods – have been employed for more than five years by the biggest software developing companies worldwide – in SMEs, too.

Static Analysis

- Tool-based code reading complements the two methods outlined above.

These are the most cost-effective methods for the successful identification of hitherto undisclosed vulnerabilities.

Further Reading

Beyond Security (Ed.): Black Box Testing. McLean 2008. http://www.beyondsecurity.com/black-box-testing.html

Beyond Security (Ed.): Beyond Security introduces 80/20 rule for 'smart' blackbox testing in new version of beSTORM. McLean 2006. http://www.beyondsecurity.com/press/2006/press12090601.html

Codenomicon (Ed.): Buzz on Fuzzing. Cupertino 2007. http://www.codenomicon.com/products/buzz-on-Fuzzing.shtml

Fox, D.: Fuzzing. DuD 30, 2006, 12, 798. 2006.

MITRE (Ed.): 2010 CWE/SANS Top 25: Focus Profiles - Automated vs. Manual Analysis. Eagle River 2010c. http://cwe.mitre.org/top25/profiles.html#ProfileDesignImp

Peter, M.; Karen, S.; Romanosky, S.: Complete Guide to Complete Guide to the Common Vulnerability Scoring System Version 2.0. Gaithersburg 2007. http://www.first.org/cvss/cvss-guide.pdf

Pohl, H.: Zur Technik der heimlichen Online-Durchsuchung. DuD 31, 9, 2007. http://www.dud.de/binary/DuD_Pohl_907.pdf

Rathaus, N.; Evron, G.: Open Source Fuzzing Tools. Amsterdam 2007.

Doyle, F.; Fly, R.; Jenik, R.; Manor, D. Miller, C.; Naveh, Y.: Open Source Fuzzing Tools. Amsterdam 2007

Pohl, H.: Entwicklungshelfer und Stresstester - Tool-gestützte Identifizierung von Sicherheitslücken in verschiedenen Stadien des Softwarelebenszyklus. In: <kes> - Die Fachzeitschrift für Informations-Sicherheit, 2, 2010. http://www.softscheck.com/publications/Pohl%20Kosteng%C3%BCnstige%20Identifizierung%20von%20Sicherheitsl%C3%BCcken%20110320%20White%20Paper%20.pdf

Schwab, F.; Findeisen, A.; Pohl, H.: Bedrohungsmodellierung (Threat Modeling) in der Softwareentwicklung. GI-Edition: Lecture Notes in Informatics. Heidelberg 2010.

Sutton, M,; Greene, A.; Amini, P.: Fuzzing - Brute Force Vulnerability Discovery. New York 2007.

Takanen, A.; Demott, J.D.; Miller, C.: Fuzzing For Software Security Testing And Quality Assurance. Norwood 2008