FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Evaluation Criteria for Security Testing Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Jasmin Adamer, BSc.
Matrikelnummer 0325259

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Thomas Grechenig

Wien, 20.09.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

.

Evaluation Criteria for Security Testing Tools

MASTER THESIS

for the obtainment of the academic degree

Diplom-Ingenieur/in

in

Software Engineering/Internet Computing

by

Jasmin Adamer, BSc.
Registration Number 0325259

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig

Vienna, 20.09.2011 __________________________ __________________________
(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluation Criteria for Security Testing Tools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Jasmin Adamer, BSc.
0325259

ausgeführt am
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Thomas Grechenig

Wien, 20.09.2011

Thesis Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

Vienna, 20.09.2011 ---

Jasmin Adamer

Development managers, the story goes, can save time, money, and aggravation by replacing pesky testers with these tools. These myths are spread by tool vendors, by executives who dont understand testing, and even by testers and test managers who should (and sometimes do) know better.

Cem Kaner

Contents

Abstract

Software testing is needed to assure quality of software, especially security. Lots of automated tools exist to ease the work of testers to find security vulnerabilities but there is no guide how to evaluate and select an appropriate tool. Every tool is supporting different features by which a random choice might not represent the most fitting tool for the software to test. Therefore, a kind of measurement to ease the selection is needed. In this thesis, evaluation criteria were defined which are based on common criteria published in scientific papers. On the basis of this, an evaluation process is worked out to support testers to find an optimal testing tool. The design of the process guarantees short training periods for the tester. The focus is on security testing tools which are specialised in fuzz testing. Fuzz testing uses invalid or wrong input to cause wrong behaviour or a crash of the application in order to reveal security vulnerabilities. Furthermore, a huge amount of random data is produced and needed for executing fuzz testing. In most cases the generation of input data is done automatically by tools. Automation is an important requirement as tools shall be able to create and execute test cases as well as compare results autonomous.

The criteria are described in detail so that they are comprehensible. In addition, for each criterion a statement how to verify its fulfillment is given. The solution is also checked out practically on the basis of four security testing tools to show how the process works. The practical part is done on Windows XP Professional. With the described evaluation criteria, every tester should be in the position to elect a suitable testing tool for successful security testing. Besides, the conducted evaluation shows that none of the selected tools is completely automated as manual input is needed to compare test results.

Keywords: Evaluation Criteria, Security Testing Tools, Evaluation Process

Zusammenfassung

Testen von Software ist heutzutage essentiell, um den Sicherheitsgrad von Software-Produkten möglichst hoch zu halten. Dafür stehen bereits diverse automatisierte Testwerkzeuge zur Verfügung, wobei jedes Stärken in bestimmten Bereichen aufweist. Durch die Vielzahl an angebotenen Werkzeugen ist es allerdings schwierig, einen Überblick über die unterschiedlichen Funktionalitäten zu erhalten. Daher wurde zunächst ein Kriterienkatalog erarbeitet, der auf definierten und in der Community publizierten Auswahlkriterien für Sicherheitswerkzeuge aufbaut. Darauf basierend wird ein Auswahlverfahren erarbeitet, das den Tester bei der Auswahl eines optimal geeigneten Testwerkzeugs unterstützt, um Sicherheitslücken rasch und effizient aufzudecken. Bei dem in der vorliegenden Arbeit konzipierten Verfahren wurde großer Wert auf kurze Einarbeitungszeiten für Tester und rasche Umsetzbarkeit der Auswahlkriterien gelegt. Der Schwerpunkt wurde auf Sicherheitswerkzeuge gesetzt, die auf fuzz testing spezialisiert sind. Beim fuzz testing werden ungültige, unerwartete oder falsche Eingabewerte verwendet und es wird eine große Menge an zufälligen Testdaten produziert. Ziel ist es, ein Fehlverhalten der Applikation beziehungsweise einen Systemabsturz zu verursachen, um Sicherheitsschwachstellen aufzudecken. Die Generierung der Daten wird meist von Sicherheitswerkzeugen automatisch vorgenommen. Weiters spielt der Punkt Automatisierung eine wichtige Rolle, da die Werekzeuge im Stande sein sollen, sowohl die Testfälle selbstständig zu erstellen und durchzuführen als auch die Ergebnisse zu analysieren und vergleichen.

Die Evaluierungskriterien werden ausführlich beschrieben, um Nachvollziehbarkeit zu erreichen. Weiters wird für jedes Kriterium angegeben, wie auf dessen Erfüllbarkeit überprüft werden kann. Der Prozess wird an vier verschiedenen Werkzeugen angewandt, um beispielhaft dessen Verwendung darzustellen. Der praktische Teil der Evaluierung wird auf dem Betriebssystem Windows XP Professional durchgeführt. Mit Hilfe der vorgestellten Kriterien soll es jedem Tester möglich sein, Sicherheitswerkzeuge zu vergleichen und ein für die bevorstehenden Tests geeignetes Werkzeug auszuwählen. Außerdem konnte gezeigt werden, daß keines der gewählten Sicherheitswerkzeuge vollautomatisiert ist, da stets manueller Eingriff notwendig ist, um die Testresultate mit den gewünschten Ergebnissen zu vergleichen.

Acknowledgements

It is with immense gratitude that I acknowledge the support of my Professor Thomas Grechenig.

This thesis would not have been possible without the support and patience of my mother, Martina Adamer, my father, Christian Adamer and my grandmother, Brigitta Hatzl. I owe my deepest gratitude to them.

I owe my thanks to my colleague, Frank Peter, who has made available his support in a number of ways.

I would also like to thank my advisor, Christian Schanes, for the help during this work.

Chapter 1

Introduction

Nowadays, computers are part of everyone's life. They are needed not only for work but also for private issues. Computer programs try more and more to ease daily life which longs for more or less complex software. Misuse, attacks or errors in software programs can lead to insecurity. The growth of software, for example web applications, gives even more possibilities for security attacks which is dangerous for sensible data of humans. The reason is that the more complex a software is, the more security leaks can exist. This is a problem which becomes more and more serious and must be handled properly somehow.

One important part of many steps to secure a system is appropriate testing. Although testing is still seen as useless and time-consuming, it makes an essential contribution to the software development process. Testing is one possibility to improve security and therefore make software useable with little risk. Successful testing means finding errors. This is especially for developers a fact which is not always accepted. Therefore a part of this thesis tries to show how important testing is by giving an overview of different testing techniques and guidelines concluding in demonstrating what advantages proper testing offers.

As soon as people accept that testing must be part of every software cycle, the question which kind of testing meets all requirements must be answered. Fuzz testing is an approach which can serve as a good starting point to detect different vulnerabilities. The goal is to reveal security leaks by testing with random and invalid data. This technique can be used in different fields of application. Four fuzzing tools, namely Fuzzolution, JBroFuzz, Peach and Fuzzware are presented in this context and evaluated in a following step.

The main contribution of this thesis is the presented evaluation criteria. To evaluate testing tools, certain criteria should be chosen to make a comparison possible. Criteria depend on what the tester wants to test and which software paradigm shall be tested. For this work, the aim was to test security testing tools which can deal with Extensible Markup Language (XML) based web applications. Therefore, some general criteria were described, like usability, SUT (software under test) access requirements and automation, which are a kind of prerequisites that must be fulfilled to put a certain tool on the shortlist. The other category, vulnerability detection criteria, are used to evaluate if the testing tool can reveal certain vulnerabilities, like SQL(Structured Query Language) Injection and cross-site scripting(XSS) in web applications.

When the user has decided which criteria must be fulfilled by a tool, the evaluation can be performed. The evaluation in this work is mainly done theoretically based on documentation and Internet research. One criterion, namely the detection of SQL Injection vulnerabilities, was evaluated using a vulnerable web application. It shall show how to deal with evaluation criteria and how to execute the evaluation. All of the four presented tools are rated and compared to figure out which security testing tool is the most suitable for web applications.

The work is structured as follows: Section 2 gives an overview of testing, explaining what it is, presenting some guidelines, showing why it is important including its advantages and shortly presenting the testing process. Chapter 3 introduces fuzz testing and presents the chosen security testing tools, Fuzzolution, JBroFuzz, Peach and Fuzzware. In Section 4 the criteria which are the basis for the evaluation are described whereas in Chapter 5 the evaluation is conducted and results are analyzed. Finally, Section 6 represents the conclusion.

Chapter 2

Testing and its Benefits

This Chapter will give an overview of testing and also explains why testing is important. Therefore, various scientific papers and books were consulted. It also includes some testing guidelines, different testing strategies which will be introduced and an overview of the test process. The main purpose of this chapter is to give a feeling of what testing is and why it is absolutely needed in every software development process.

2.1 Testing in General

Testing is part of every software development process in different measures. It gets more and more important as it is responsible for more than half of the costs of software as also stated by Jones and Chatmon [1]. Most companies have separate internal or external testing departments or at least some testers in the development team to cover this effort. In general, testing sounds easier than it is in reality. To show the complexity of the process of testing, a definition of what testing is and what the test process includes is needed.

Boris Beizer [2] defined testing as follows: "**Testing** is the act of executing tests. Tests are designed and then executed to demonstrate correspondence between an element and its specification". Therefore, testing is useful and definitely needed for every software program. The need of testing is approved in "Software Test Automation - Effective Use of Test Execution Tools" [3] where the authors emphasize that testing must be both, efficient and effective. The goal is to detect vulnerabilities fast and to convince the user to trust the software. This means that the goal of testing is to detect as many security gaps as possible. In "Testing in the 'small'" [4] the target of testing is defined more precise by stating that verification and validation are the main parts of it. So testing turns out to be extremely time consuming and thus is often declared as a useless, expensive task.

Summarised, testing is basically a process to identify errors or wrong behaviour of software. Thus, the process of testing is an indispensable contribution to the quality of software as also written by Antonia Bertolino [5] for whom testing should fulfil two tasks:

- Fixing of errors
- Quality assurance, like reliability and usability

Another significant goal of testing is to ensure security. As the size of software projects grows, and also the purpose changes a lot, it is not easy to ensure secure programs. Nevertheless, security issues must be emphasized especially for software projects which handle sensitive data or are suited to jeopardize human life. This can be realized through security testing which stands for the following: "The phrases *security testing*, *penetration testing*, and *red-teaming* have traditionally referred to executing a suite of scripted tests that represent known exploits." [6]. The fact that lots of different security attacks with diverse complexity exist nowadays makes security testing difficult and challenging as also stated by Herbert H. Thompson in "Why Security Testing Is Hard" [6] where he compares the work of testers to the work of detectives. In Figure 2.1 the circle demonstrates how the software should work. In contrast, the shapeless part shows the implemented functionality. Lots of vulnerabilities can be found in the red part of the Figure as side effects [6].

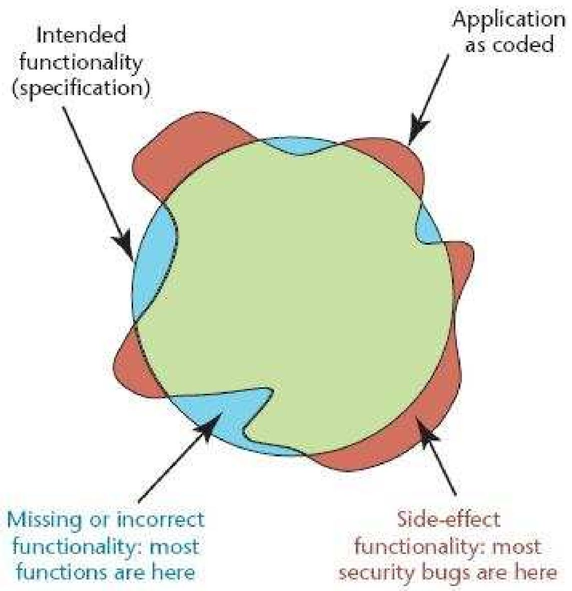

Intended functionality (specification)
Application as coded
Missing or incorrect functionality: most functions are here
Side-effect functionality: most security bugs are here

Figure 2.1: How software should work and how it works in reality [6]

Thus, the author tried to find a suitable testing technique to solve this problem with the result that there is no technique at the moment which is able to reveal a huge amount of security issues. So the question how to test for it remains [6]. This leads to the fact that security testing needs extremely efficient tools which are able to deal with the characteristics of security leaks as noticed by Herbert H. Thompson [6].

Consequently, security testers must meet various conditions to be qualified for their job. This is shown in Figure 2.2 by Matthew Nicolas Kreeger [7] who also stated that a security tester needs experience in:

- Programming
- Test creation
- Security
- Cryptography

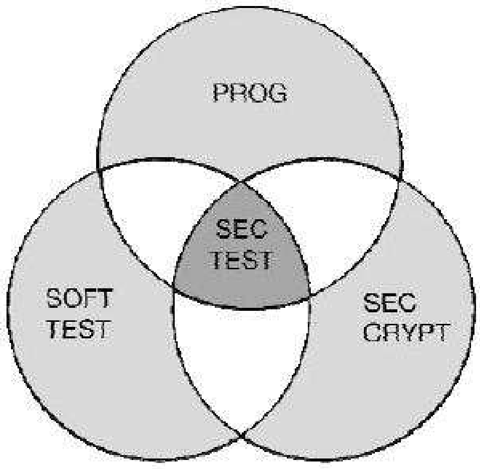


Figure 2.2: Capabilities which a software tester should have [7]

2.1.1 Testing Guidelines

Basic suggestions are existing which shall be kept in mind when testing software to offer efficient testing. Depending on the type of software paradigm, there are even more precise ones. Some of the best practices which can be found in every software process literature are presented here [2] [5] [8] [9] [10] [11] [12] [13] [14] [15] [16]:

- Testing means showing that there are errors

  Every tester and also all other people involved in the development process must keep in mind that testing means proving that errors exist. A good tester is a person who identifies a wrong behaviour of the software and not one who proves that there are no errors. The more malfunctions found, the better it is. This is also stated by Boris Beizer: "Not only are all known approaches to absolute demonstration of program correctness impractical, but they are impossible as well. Therefore, the objective of testing must shift from an absolute proof to a suitably convincing demonstration; from a deduction to a seduction; to the creation of warm feeling in everybody's tummy instead of heartburn. What constitutes a "suitable" demonstration depends on context." [2]. Even more concrete is the statement of Pamela R. Pfau [15] who says that although testing is needed to show that a product works as it is specified, the testing technique itself needs to lead to a wrong behaviour of the software. This would then be put on a level with success. Otherwise, if everything works perfectly, the tester might not have been successful or at least will not have a sense of achievement.

- Do not test your own code - one role per person

  No developer should test his or her own code and no tester should programme which means nobody should have both, the role of the developer and the role of the tester. The reason for this is quite simple: Everybody believes in oneself and is convinced, that he or she does not blunder. This stands in conflict with the purpose of the tester role. If someone feels confident that his or her work is accurate, the person will hardly find errors as objectivity is not given. Boris Beizer states furthermore that there should be also a clear separation between the test designer and the test executor, meaning that the persons who design the test cases should not be the same as those who execute it. In addition, a test designer should do his work with such a quality that every competent person is able to execute the tests. The test executer should execute tests autonomously which means without the help of the designer. This reveals discrepancies in test design and documentation [2].

  Not only dispassion but also the fact that it is always easier to reveal mistakes of other people than those made by oneself leads to the recommendation of different persons per role. Another point is that two heads are better than one. It is impossible to think of every potential input or action a future user can do, but the more persons that are involved, the more different scenarios will be thought of. In addition, it is more effective to concentrate on one role. In contrast, Tim A.

Majchrzak [12] stated that it is not absolutely necessary that a person fulfils only one role. The point is that a person needs to concentrate only on one role and its duties at a time. This does not preclude being both, developer and tester.

- Testers must be involved through the whole software life cycle

  To guarantee efficient and complete testing, testers must be integrated from the beginning to the end of the software process. This is the only way to make sure that testers know which functionalities the software shall feature. This best practice is also mentioned by Elfriede Dustin [9] who says that testers have to be part of the project's life cycle from scratch to be able to understand the software and test properly. Furthermore, it eases testing if a tester can communicate with stakeholders for establishing requirements.

- There is no bug-free code

  People make mistakes and also developers, no matter how experienced and educated they are, are not safe from errors. So although it is a wish of every programmer, code is never bug-free. This indicates also that, no matter how huge the application is and how good and qualified the tester is, the software will contain errors after roll-out at the client. This point is closely linked to the fact that every individual has a different perception and as a result the way software is used varies which might lead to an untested behaviour of the program. That bug-free code is an illusion is also stated by Boris Beizer when he says that the goal of showing that software is bug-free is simply practically and theoretically impossible [2].

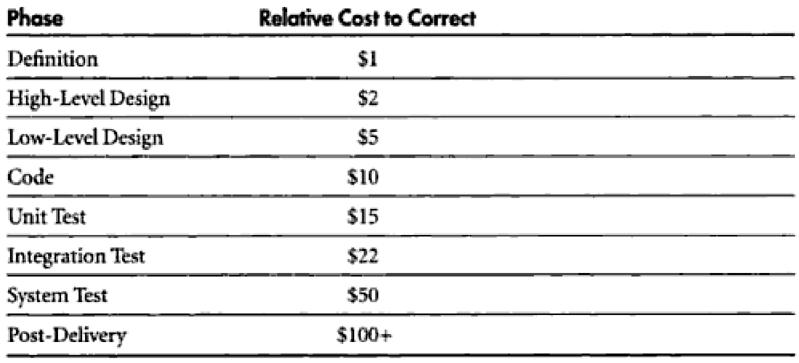
- Testing is part of the whole development process

  Testing is not a single phase which can be done at the end of the development process only. Instead, testing is an iterative process which must be included from the beginning of the software cycle as also described in "Testing E-Commerce Systems: A Practical Guide" [10] where the author states that testing has an important role in the whole development process. Lots of arguments for such a strategy exist:

  Time An error, which is found at the end of the development process, costs a lot of time because it is hard to find the responsible lines of code in a huge amount of classes. Antonia Bertolino stated in "Software testing research and practice" [5] that with testing failures can be detected. Nevertheless, analysis which might be cost-intensive because of the size of software is needed to find the source of those failures. Furthermore, even a small change of a part of the code might affect the rest of the program and lead to side effects. If the complete software is based on an error-prone foundation, the project is at risk. This fact is confirmed in "Testing E-Commerce Systems: A Practical Guide" [10] where the author comes to the conclusion that huge defects which are found during the late phases of the process lead to time pressure regarding finding a solution for the problem.

Effort Searching through thousand lines of code leads to enormous effort on the developer's side. He or she has to figure out the root of the fault and, additionally, the tester has to repeat all tests resting upon the defective code which might be the complete software.

Money Both, the time and the effort factor result in a lack of money. Testing in general, searching for and fixing problems and also repeating tests afterwards is at great cost as also stated in "Automated testing from specifications" [8]. The authors point out that especially for safety-critical software the verification and validation part is the main contribution to costs. The more code to be analyzed and the more testing cases to be recapped, the more dearly it is. This can have fatal consequences, particularly a considerable loss of money or even a business with a deficit, respectively a stop of the project in all. Figure 2.3 shows that costs are closely related to time as the expense of a bug multiplies in later phases.



Phase	Relative Cost to Correct
Definition	$1
High-Level Design	$2
Low-Level Design	$5
Code	$10
Unit Test	$15
Integration Test	$22
System Test	$50
Post-Delivery	$100+

Figure 2.3: Increase in cost of bugs during different phases in the development process as illustrated by B. Littlewood [14] in Software Reliability: Achievement and Assessment(as cited in [9], 2002, p. 4)

All in all, testing must be part of every step of the software development process and should start as soon as possible, for example starting test specification during the elaboration phase and testing every class

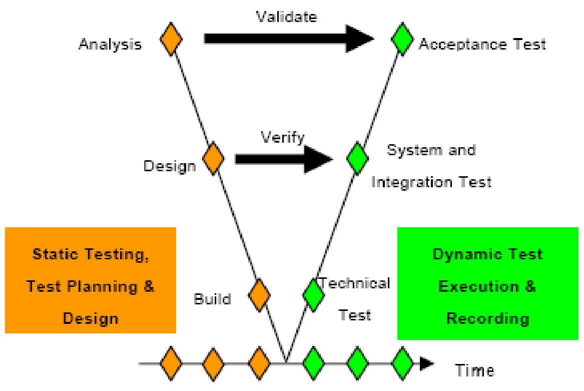


Figure 2.4: How the V-Model looks like [11]

or module immediately, even if it will be extended later. The earlier a defect is detected, the better it is for the project. Therefore, Londesbrough integrated different testing phases into the V model as can be seen in Figure 2.4 whereby the diamonds epitomise deliverables and milestones, like test cases generated in each phase [11].

Furthermore, testing needs to be planned like every other field of the development process as also stated by Londesbrough [11] who showed that a proper, coordinated test process which attend the whole project from the beginning to the end is needed for the success of the software. It makes no sense to test software uncoordinated.

- Be as precise as possible

  To facilitate the reproducibility of test cases with a different outcome as expected, the tester must record the test case as also commented in "Workshop Report: Software Testing and Test Documentation" [13] where it is stated that the decided testing method is not linked to the need of documentation of tests. The description must be clear and include all necessary details, like environment, list of actions, inputs, real and expected outcome, to give the developer the possibility to duplicate the incident immediately without that asking for additional information is needed. This saves a lot of time and effort when fixing bugs.

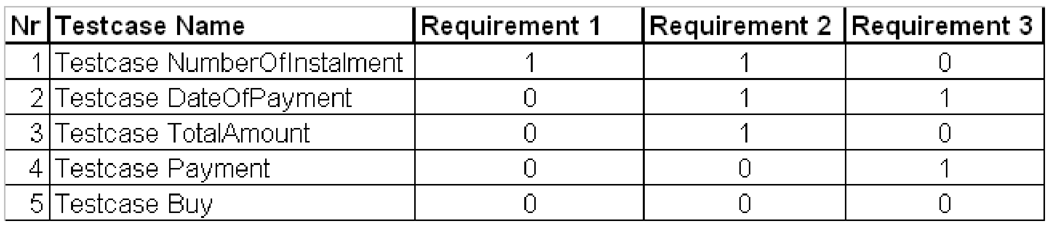
- Documentation of the test process

  The whole testing process must be recorded. Efficient testing of the application shall be the intention. The documentation must at least include the name of the tester(s), dates, application name, testing strategy, test cases including id, name, performer, expected and actual result. The tracing of the whole test process has to be possible.

E. Miller [13] explained the importance of documentation. The author explained that documentation is needed to have some written agreement on the different opinions of developers, testers and users. This has nothing to do with the chosen test technique. For software which might jeopardize human life it is even more important to have the testing process documented as this is needed for the regulatory agencies. Furthermore, it serves as basis for the judgement of such agencies.

- Keep track of requirements

  Requirements succumb changes constantly during the software development process. Hence, it must be ensured that they are throughout up-to-date. A good way to keep track of requirements is to save them in a database as it facilitates updates. Another possibility is to create a matrix such as Figure 2.5 whereby "1" stands for fulfilled and "0" for not fulfilled. This approach makes sure that the test cases cover all requirements.



Nr	Testcase Name	Requirement 1	Requirement 2	Requirement 3
1	Testcase NumberOfInstalment	1	1	0
2	Testcase DateOfPayment	0	1	1
3	Testcase TotalAmount	0	1	0
4	Testcase Payment	0	0	1
5	Testcase Buy	0	0	0

Figure 2.5: Possible illustration of the relation between requirements and test cases

  Of course, it should be attempted that the information which test case fulfils which requirement is also in the database. This best practice is explained in "Increasing Understanding of the Modern Testing Perspective in Software Product Development Projects" [16] where it is handled under the point "traceability and maintainability": "*Traceability and maintainability* includes ways of connecting testing to requirements as well as considerations on the ability to maintain and grow the tests. Test cases should be grouped into test suites of different priorities, different functionalities and different uses (e.g. smoke test, regression test) to facilitate control. A traceability matrix between the test cases and requirements should be kept up-to-date in order to know if tests need to be updated, as well as what tests need to be updated, to the changing requirements.".

2.1.2 Testing Strategy

Different strategies of how to test software exist whereas some of them need more details about the program than others. A combination of all test levels is necessary to guarantee a good quality at all layers.

Black box vs. White box vs. Grey box

One possibility to divide testing is to split it up into black box and white box testing. Black box testing means simple input - output testing to guarantee the satisfaction of functional requirements whereas the code is irrelevant for the tests and therefore the tester does not need to have access to it. In "Coverage metrics for requirements-based testing" [17] black box tests are defined as tests where the implementation details are not interesting at all. Instead, the requirements are taken as basis for establishing the tests. The fact that the output, or more precise the behaviour of the software, is most important leads to the need of a complete and accurate requirements specification. Otherwise black box testing is useless.

The opposite is white box testing where tests are based on the code. In "Reliability-oriented Software Engineering: Design, Testing and Evaluation Techniques" [18] this test strategy is described in more detail. The author describes white box testing which is also called coverage testing as a technique to evaluate the testing quality on the basis of the structure. White box testing is only possible if the tester has full access to the code.

Summarized, the authors of "Software unit test coverage and adequacy" [19] define the difference between the two testing strategies whereby for black box testing no information about the source code is necessary. In contrast, white box testing is based on implementation details which must be available to the tester.

Grey box testing is situated in-between and kind of a mix of black box and white box testing. Grey box testing becomes more and more popular as testing strategy. For this strategy, some information about the implementation of the software is needed as stated by Baharom and Shukur in "Module Documentation Based Testing using Grey-Box Approach" [20]. So, parts of the code need to be accessible by the tester for the grey box testing strategy.

Levels of Testing

Testing can take place on different levels according to the presented V-model which leads to a split of testing strategies, for example into unit testing, integration testing, system testing, acceptance testing and regression testing.

Unit testing means that a single unit of the program is tested. Antonia Bertolino [21] states that unit testing is one of the most important test phase for quality assurance as it is able to find errors which are deeply-hidden. Those faults might not be detected in system testing. Additionally, Boris Beizer [2] explains that "unit testing is a rite of passage in which the unit is transformed from the private to the public domain.".

The *integration test* is important to guarantee that all parts of the systems are compatible, as also described by Antonia Bertolino [5] in "Software testing research and practice". The author states that with integration testing the cooperation between subsystems is examined.

Regression tests consider side effects and compare the result with the outcome of previous tests. Antonia Bertolino [5] defines regression tests as "test execution and re-execution". Sami Beydeda and Volker Gruhn [22] define regression testing as tests where changes are checked if they correspond to the requirements. Furthermore, it must be examined if the changes have negative effects of other software parts.

The *system test* is one of the last huge tests which must be done to ensure that the software meets the requirements. The author of "Building awareness of system testing issues" explains that system tests have the duty to find problems which were not taken into account during development testing. After the system test, those problems should be solved and the software should be better than before the test. With this test, the unwanted surprises on user's side shall be narrowed [23].

The system test includes, according to Boris Beizer [2], various tests, like stress testing, load and performance testing and configuration testing, although besides system-level functional testing, formal acceptance testing and stress testing not all of them are needed for every system. This indicates, that system tests are enormously time-consuming, but due to their importance imperative.

The *acceptance test* is the final test where the client is involved and decides whether to accept the software or not. Antonia Bertolino states in "Software Testing Research: Achievements, Challenges, Dreams" [21] that acceptance tests are used for validating large software products on client's side.

In one of the standards of the Institute of Electrical and Electronics Engineers(IEEE) which gives guidelines to conform to Portable Operating System Interface for Unix(POSIX(R)), namely the "IEEE Standard for Information Technology Requirements and Guidelines for Test Methods Specifications and Test Method Implementations for Measuring Conformance to POSIX(R) Standards" [24], a different allocation of testing levels is given. According to the standard, testing can be divided in exhaustive, thorough and identification testing.

- Exhaustive testing: Exhaustive testing is defined as testing the attitude of each aspect that an element exhibits whereby all permutations must be taken into account. This means that exhaustive testing includes testing with no input string, one single digit and so on when testing an input field which accepts up to three digits at once [24].

- Thorough testing: A bit different is thorough testing. Basically, it is completely the same as exhaustive testing with the restriction that not all permutations are taken into consideration [24].

- Identification testing: In contrast to this stands identification testing where the goal is to test for certain characteristics of the test element. These tests do not go into detail as only the minimal functionality should be approved [24].

All in all, this distinction of testing levels plays an important role considering the necessary time for performing tests as more tests are more time-consuming. In general, the testing level is important regarding the choice of testing tools as different preconditions must be given, for example for white box testing or code-based testing it is indispensable to have access to the code.

2.2 Manual vs. Automated Testing

Manual testing means that the tester is executing each test case manually. In "Reconciling Manual and Automated Testing: The AutoTest Experience" [25] the authors state that this is customary technique where testers establish test cases which will best fit to the execution of the program in their opinion. Basically, manual testing is splitted into "Code based testing" and "Specification based testing" according to Srinivasan and Leveson [8]. In Figure 2.6 of "Automated testing from specifications" [8] the difference is shown graphically.

- Code based testing

Code based testing is code orientated which means the tester needs access to the internal structure and the code is already existing, developed on basis of a given specification [8].

- Specification based testing

Specification based means that the test cases are generated by the specification which presupposes a complete and clear specification. In the Figure, it can be seen that tests, program and pass/fail criteria are based on the specification. If the specification is executable, its outcome can be collated with the results of the tests which were executed on the program [8].

Nowadays, it is nearly impossible to guarantee quality with manual testing only as applications are too comprehensive and manual written test cases can simply not cover all problem areas, as also stated by the authors of "Predictive testing: amplifying the effectiveness of software testing"

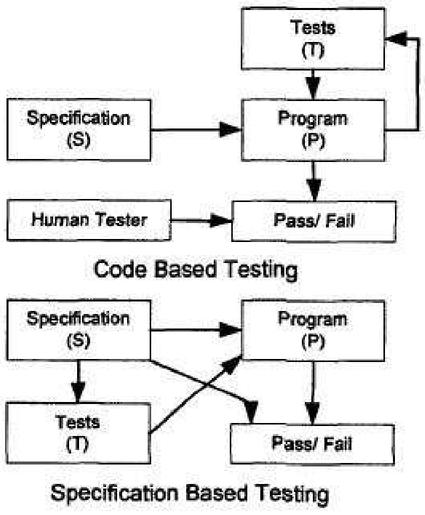


Figure 2.6: Structure of manual testing types [8]

[26]:"Unfortunately, testing using manually generated test inputs often results in poor coverage and fails to find many corner case bugs and security vulnerabilities resulting from buffer overflows, integer overflows, etc.".

Furthermore, executing each test case manually is extremely time-consuming. This leads to a need for a suitable automated testing tool. The author of "Improving Effectiveness of Automated Software Testing in the Absence of Specifications" [27] comments the derived demand for such tools with the reason that manual testing is quite tedious. Therefore, automated testing tools can be used by testers to at least ease some of the parts of testing. The authors of "Observations and lessons learned from automated testing" [28] share this opinion. They state that for keeping the effectiveness of testing automated testing tools need to be applied which execute substantial tests like load and performance tests. Nevertheless, automated testing has challenges for testers too, those are "generate test inputs effectively" and "verify test executions effectively" according to Tao Xie in "Improving Effectiveness of Automated Software Testing in the Absence of Specifications" [27]. This shows, that despite testing tools, human testers are needed although often declared as useless as explained by Cem Kaner in "Pitfalls and Strategies in Automated Testing" [29]: "Development managers, the story goes, can save time, money, and aggravation by replacing pesky testers with these tools. These myths are spread by tool vendors, by executives who don't understand testing, and even by testers and test managers who should (and sometimes do) know better."

A good comparison of manual and automated testing can be found in "Reconciling Manual and Automated Testing: The AutoTest Experience"

[25]. For testing in the deep manual tests are more suitable than automated ones. On the other side, to reach a good coverage, automated tests are needed as the huge amount of test cases cannot be handled manually. In "Reconciling Manual and Automated Testing: The AutoTest Experience" [25] the authors also explain why manual testing is not dispensable. This is for the simple reason that humans are more efficient at unit testing because of generating test cases which might be more suitable to detect errors. In addition, testers are better at constructing sophisticated input.

All in all, a good testing process includes both, manual and automated testing to exhibit a satisfactory result.

2.3 Benefits of Testing

Testing provides a lot of advantages which are often not considered at all and is able to reduce risks. Thus, it is indispensable regardless of the size of a software project. Nevertheless, testing is often considered as useless, time consuming and expensive, as also stated by Boris Beizer in "Software System Testing and Quality Assurance" [2]: "Testing is often conducted as a meaningless, objectionable ritual that proves nothing, demonstrates less, and is more likely to create dissension than quality.". In reality, a huge variety of leverages can be gained through testing. Main benefits are among other things:

- Time, Cost and Effort: Although testing brings a lot of effort, time and cost consumption with it, it also reduces the time exposure, expenditure and effort of searching bugs in the whole code at the end by exposing errors early in the test process. This benefit is also outlined by Amitabh Srivastava and Jay Thiagarajan [30] who see the main duty of testing in find errors whereby if this happens early in the process, the effort for the factors time and resource is reduced. Nevertheless, testing causes high costs.

- Quality: Each project must offer a certain quality otherwise it will not be accepted by the customer. Poor quality leads to claims and additional work for the company. Besides, it influences the reputation of the enterprise and therefore the order position might suffer. Proper and extensive testing according to a defined test process is the only way to assure the quality of software. That testing is essential for this issue is known for many years now as can be seen in the statement from Pamela R. Pfau [15] of 1978. The author says that quality assurance has the duty to judge software with the aid of macroscopic testing. How macroscopic testing looks can be seen in Figure 2.7.

- Meeting of the requirements: Every project has risks. Testing can reduce some of them, especially the risk of not meeting the requirements

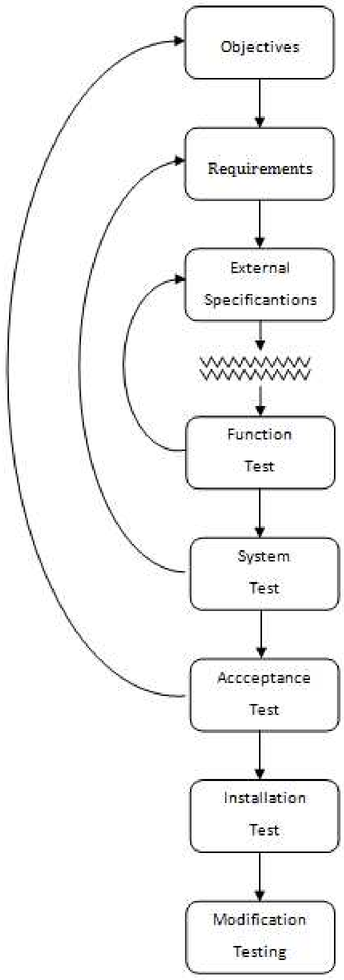


Figure 2.7: Process of macroscopic testing [15]

as explained in "The Testing Process - A Decision Based Approach" [31]. In this paper, the authors state that testing is needed to find out if the software meets the requirements. This is important as nowadays the desired functionality grows more and more with simultaneous reduction of time and resources. So it is easy to figure out if the software satisfies the needs of the customer by testing on the basis of the specification. Testing is needed to make sure that the scheme complies with the client's standard.

- Satisfaction of clients: The most important goal of a company should be the satisfaction of clients as they are responsible for the success of the enterprise. If software does not meet the requirements or is extremely error-prone, the customer will not accept it which leads to a huge loss for the company. Furthermore, the customer will search other contractors for their projects. To satisfy clients, not only price and time is important but also the quality must be extremely good. So testing can give another valuable input to the software development process by assuring quality and showing customers via user acceptance tests that requirements are met.
- Security: With the increase in applications which deal with confidential data, security must be guaranteed. This can be realized through proper testing. How important testing in the context of security is, is stated in "Guidelines for secure software development" [32]. The authors confirm that a good security strategy must incorporate testing. Additionally, it is necessary that developers stick to standards to implement less security gaps and that the test process uses testing standards and best practices.

  So with the correct testing approach, security issues can be revealed and fixed by the developers without any risk.

Summarized, testing is important as it is a way to achieve and verify quality. If the customer feels that the product offers bad quality, a takeover of the software will not take place and this will result in a huge loss for a company and a bad reputation. Such a scenario can be prevented by proper testing. Unfortunately, the acceptance of testing is still missing although the advantages have been clear for years.

2.4 Test Process and its Place in the Software Development Cycle

This Section shall give an overview of the test process and also explain where testing is situated within the software development process.

2.4.1 The Place of Testing

As testing is part of every development process, it is important to know where it is situated in the project life cycle. A common procedure model is the waterfall model where testing starts at the end of the project as can be seen in Figure 2.8.

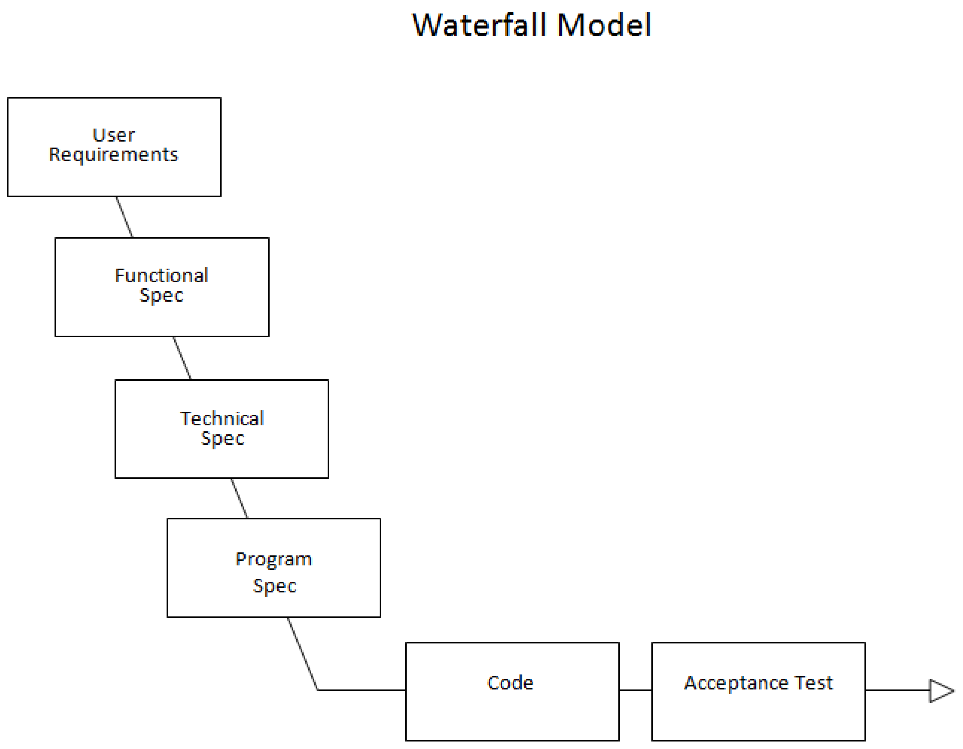


Figure 2.8: The place of testing in the waterfall model [33]

In contrast, the V-model exists which should be used nowadays as it fits more the requirements for modern software life cycles. Testing is part of the whole process in the V-model as shown in Figure 2.9. Graham Davis [33] sees the advantages of this model as follows whereby effectiveness comes before efficiency:

- Effective implementation because of the early existence of test cases
- More efficient testing as automated tools can be fielded

2.4.2 Test Process

Testing is needed to guarantee quality but useless if it is not executed within a certain process. To perform prosperous testing, specific measures must be

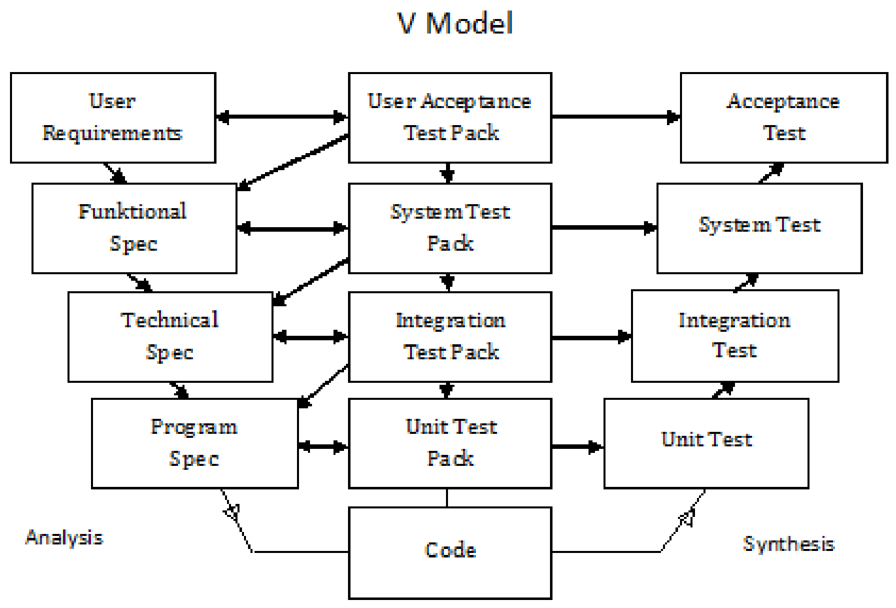


Figure 2.9: The place of testing in the V-model [33]

adopted. A test process which requires documents, plans and other issues has to be established. In Figure 2.10 an example of a test process can be seen.

The process has to be iterative as testing against requirements can only take place if the requirements are kept up-to-date during the whole life cycle. In conclusion, retesting is also necessary. Furthermore, not all tests will give the desired result or test cases will be wrong formulated by which the responsible people have to step back to an earlier phase of the process or restart it from the beginning.

The aim of the testing process is described by Anne Mette and Jonassen Hass [34] as offering information which is needed to keep the quality of products, decisions and processes in the next testing employment.

Every good test process consists of basic inputs, activities and outputs [34] whereby the inputs are the following:

- Strategy for testing
- Project plan
- Master Test Plan (MTP)
- Details about the test progress

The activities include several tasks as listed below:

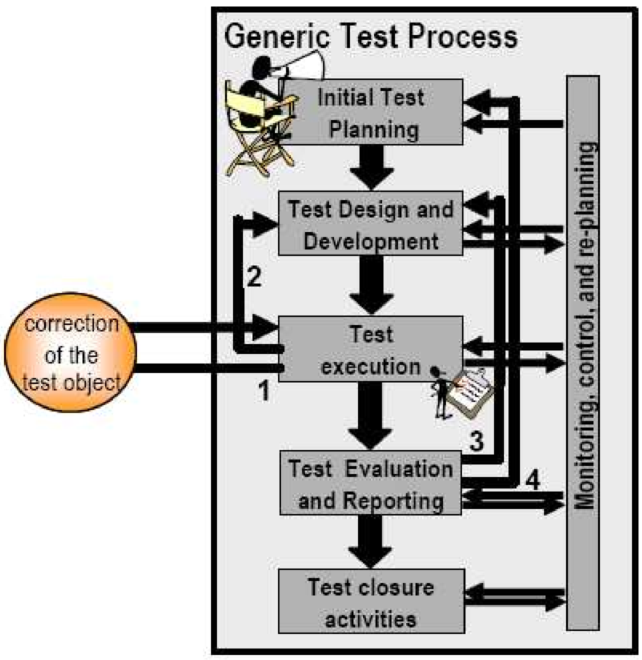

Generic Test Process
Initial Test Planning
Test Design and Development
Test execution
Test Evaluation and Reporting
Test closure activities
Monitoring, control, and re-planning
correction of the test object
1
2
3
4

Figure 2.10: Example of an iterative generic test process [34]

- Planning the test at the beginning
- Monitoring, controlling and new planning
- Designing and implementing of tests
- Performing the tests
- Analysis of the test and corresponding reporting
- Close-down activities

The results of the test process are in form of the following outputs:

- Specific plan for testing
- Specification of the test
- Specification of the environment for the test
- Current test environment and test cases
- Logs of the test

- Reports showing the progress
- Report that summarises the test
- Report that describes the experience during the test

Some of the necessary inputs, activities and outputs will be shortly explained to give a brief glimpse into the testing process.

Master Test Plan

A master test plan is the basis for a structured test process and is therefore crucial for every software process. According to the IEEE Standard 829-2008 [35] the Master Test Plan(MTP) fulfils several tasks:

- Offering a document containing a test plan
- Offering a document containing a test management

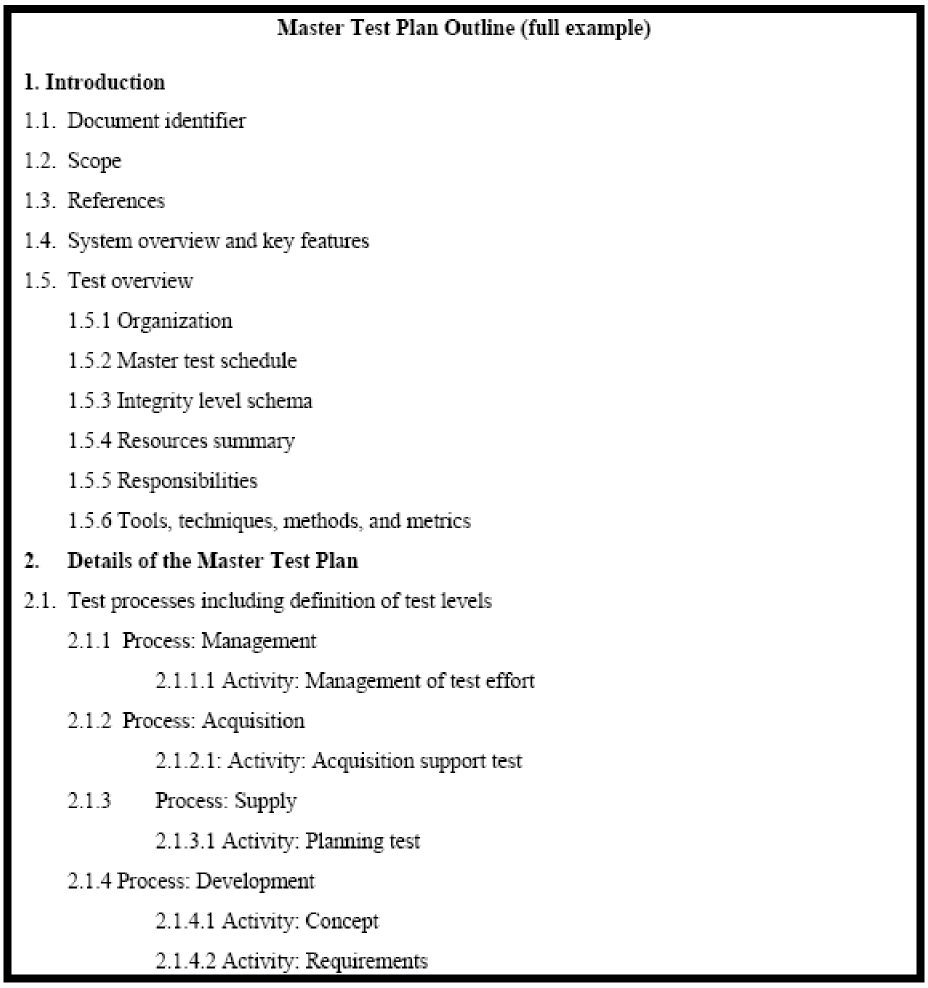
It includes activities like defining goals and correlations, establishing assignments and documentation guidelines and much more as described in detail by the IEEE standard [35]. The IEEE Standard also gives an exemplarily structure, shown in Figure 2.11 and Figure 2.12, which the master test plan should follow.

Besides the master test plan other test plans for every testing phase like for the User Acceptance Test(UAT) exist. Those are of special interest for the user as stated in "Generating User Acceptance Test Plans from Test Cases" [36]. A UAT test plan is needed to show that the implemented system correspond to the requirements. Essential scenarios should be considered in such plans.

In general, the effort for creating a test plan should not be underestimated as it grows with the progress of technology. This progress is also responsible for the demand for extended and adapted test plans as noticed by the authors of "Building a verification test plan: trading brute force for finesse" [37]. All in all test plans are needed to ensure effective and efficient testing.

Test Design and Development

This is the essential part of the test process as within this activity the test cases are written. The aim of this "phase" is as described by Anne Mette and Jonassen Hass [34] the designing and writing of test cases. Those test cases need to offer high coverage. In addition, exact information about the test environment must be provided. Still the question what a test case is exactly is open. This is answered by Mike Smith and Neil Thompson [38] who define test cases as follows:



Master Test Plan Outline (full example)

1. Introduction

1.1. Document identifier

1.2. Scope

1.3. References

1.4. System overview and key features

1.5. Test overview

1.5.1 Organization

1.5.2 Master test schedule

1.5.3 Integrity level schema

1.5.4 Resources summary

1.5.5 Responsibilities

1.5.6 Tools, techniques, methods, and metrics

2. Details of the Master Test Plan

2.1. Test processes including definition of test levels

2.1.1 Process: Management

2.1.1.1 Activity: Management of test effort

2.1.2 Process: Acquisition

2.1.2.1: Activity: Acquisition support test

2.1.3 Process: Supply

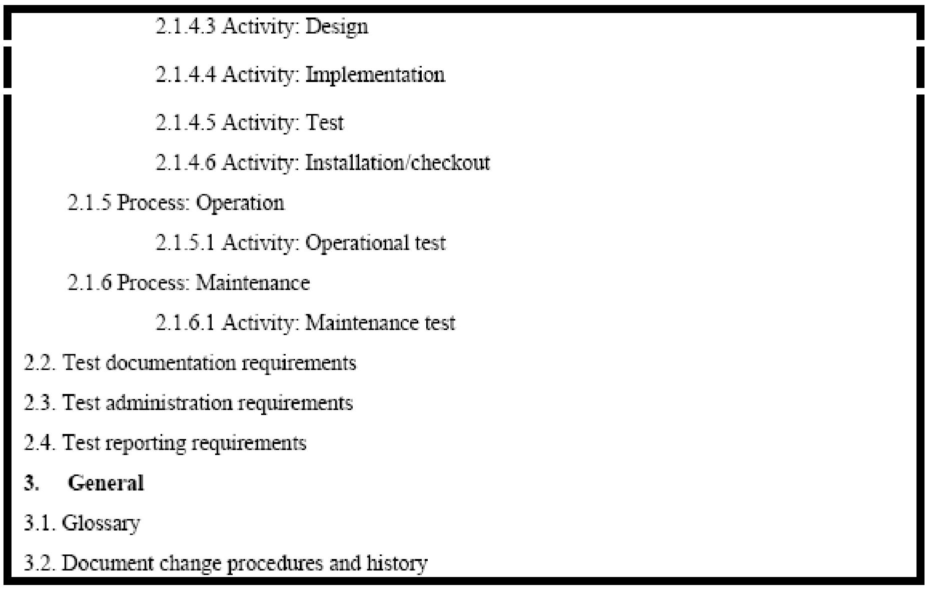
2.1.3.1 Activity: Planning test

2.1.4 Process: Development

2.1.4.1 Activity: Concept

2.1.4.2 Activity: Requirements

Figure 2.11: Possible structure of a master test plan - part 1 [35]



2.1.4.3 Activity: Design

2.1.4.4 Activity: Implementation

2.1.4.5 Activity: Test

2.1.4.6 Activity: Installation/checkout

2.1.5 Process: Operation

2.1.5.1 Activity: Operational test

2.1.6 Process: Maintenance

2.1.6.1 Activity: Maintenance test

2.2. Test documentation requirements

2.3. Test administration requirements

2.4. Test reporting requirements

3. General

3.1. Glossary

3.2. Document change procedures and history

Figure 2.12: Possible structure of a master test plan - part 2 [35]

- include the what and how
- describe data and conditions
- Can include information about requirements for the environment and procedure and the test objective

Monitoring, Control, and Re-planning

Monitoring and Controlling the test process is important and needed to check if everything is done as supposed. Besides, this is the only way to be able to react on unpredictable incidents in time. Re-planning might be necessary to keep the process up-to-date, especially if the time plan or requirements change which is in nearly every project the case. The controlling task is quite challenging as described in ”Modeling and controlling the software test process” [39] where the author states that most of the times a development process is more a creative than a physical issue which leads to an attitude of software that cannot be forecasted.

Test Logs

Test logs are necessary to make test runs comprehensible for others. Such a test log must according to Graham Davis [33] at least consist of:

- Time designation: when did the execution take place?

- Success story: did the script pass or fail?
- Information about the further course of action: must the script be executed again?
- References: where can the problem reports be found?

With this information, everyone should be able to judge tests and follow them up. It also eases the work of fixing problems for developers. In addition, members of the testing team might change during the project life cycle which makes test logs indispensable to correctly proceed with the test process. Testing is only worthwhile if a process is followed so that tasks are clear and applicable results arise. This approach is needed to ensure quality and give a good feeling at the customer.

Chapter 3

Security Testing Tools

This Chapter gives an overview of a certain type of testing called fuzz testing. The purpose is to explain what fuzzing is and present some exemplary approaches for fuzzing tools. Furthermore, it introduces the chosen security testing tools which will be used for evaluation in a later chapter.

3.1 Fuzz Testing

Testing is essential for web services as also stated by the authors of "Test-Data Generation for Web Services Based on Contract Mutation" [40] who state that testing is amongst other techniques needed to enable reusability and reliability of web services. The quality of those services is from their point of view the key for successful web services. Therefore, it is necessary to use sufficient testing techniques and tools to guarantee good quality. One possible approach to ease especially security testing for web services is to use fuzz testing. Fuzz testing is a testing technique which shows similarities with robustness testing and negative testing. With the fuzz approach, testers try to figure out the quality of a system by giving wrong or unexpected input. This kind of input is often forgotten when developing a system. Thus, the fuzz testing approach is useful to show how the system reacts on malformed input data. The authors of "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection" [41] explain the simplicity of the fuzzing process:

1. Create invalid, malicious or malformed input
2. Give the input to the application
3. Monitor the applications' behaviour
4. A crashed or a hanging application might indicate a vulnerability

One reason which stands for the fuzz approach is stated by Patrice Godefroid. He mentions that this technique is effective for revealing security gaps [42]. Still the question when a fuzzer can be defined as useful is open. Robert Brummayer and Matti Järvisalo answered this question with the simple statement that fuzzers are useful if they are able to generate a huge amount of various data. This will lead to inputs which are able to detect a lot of vulnerabilities in software [43]. The principle of fuzz testing is based on random data. Random data means that the inputs are generated arbitrary. There is no system behind the generation, the inputs are independent of each other. Random data does not follow a certain order. The success of fuzz testing is amongst other things ascribed to the way how test data is generated. That the creation of input data plays an important role for web services is also noticed by Y. Jiang, Y. Li, S. Hou and L. Zhang [40] as according to the authors the quality of such data is linked to effectiveness and costs. As a consequence, the authors even used random data for creating input for web services. Contract-based mutation testing serves as basis for their approach to automatically create data [40]. Basically, there are two approaches for generating test data for the fuzz technique, mutation-based and generation-based.

- Mutation-based Fuzzing

According to the authors of "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection" [41] mutation-based fuzzing alters test cases which consist of correct test data. This will lead to problems as soon as checksum checks are implemented.

- Generation-based Fuzzing

Generation-based fuzzing is fuzzing based on format specifications. Those specifications are used to generate data. This technique can be cost-intensive if the needed production rules must be generated based on badly documented specifications or without access to the source code [41].

This distinction can also be found in the book "Fuzzing: Brute Force Vulnerability Discovery" where the main difference between those two methods is stated as the fact that for mutation-based fuzzing existing data is manipulated and for generation-based fuzzing new input is generated [44].

Fuzzing has a big advantage when it comes to false positives. This special aspect of the fuzzing technique is described by Noam Rathaus and Gadi Evron. The authors state that fuzzers do not find false positives as every error which occurs is either a security vulnerability or jeopardizes the software stability [45].

All in all, fuzzing seems to be an excellent alternative to traditional testing methods for revealing vulnerabilities in software programs. Especially in the field of security testing can the fuzzing technique reach excellent results.

3.1.1 Fuzzing Types

Black box fuzzing is the original fuzzing technique where inputs are taken and randomly altered to get new test data. It is possible to take application-specific grammar into account and even add test heuristics through the usage of grammars for generating input as stated by P. Godefroid [42]. Another interesting point is that for black box fuzzing no access to the code is needed. This means that the tester does not have knowledge about the internal structure of the program.

In white box fuzzing, tests are generated dynamically. The authors of "Grammar-based whitebox fuzzing" [46] describe white box fuzzing as a process consisting of the following steps:

1. Using a concretely and symbolically correct input for the first run of the program
2. Symbolic execution construct constraints during this run which show how the application deals with input
3. Every constraint is negated to obtain new input data
4. The process is reprised with the new data
5. As a result, lots of control paths shall be passed to find vulnerabilities

For white box fuzzing, access to the source code is needed. The chosen fuzz type depends on the situation meaning which kind of software needs to be tested as well as on the access rights of the tester.

Besides the main types, black box and white box fuzzing, several extensions of the basic types exist. S. Sparks, S. Embleton, R. Cunningham and C. Zou, for example, extended the black box fuzzing approach based on Dynamic Markov Model fitness heuristic to enable testing for vulnerabilities at certain points in the control flow graph. Therefore, the transition behaviour of the input data is seen as roughed and on the basis of this an absorbing Markov process which is a probability model is created [47]. The probability values are estimated on the basis of tested inputs representing examples of the Markov model. Furthermore, a context-free grammar is used to define the input structure and a special genetic algorithm, grammatical evolution, is needed to produce strings. This creation is handled by a genome whereby a genome stands for a possible solution of a problem in a genetic algorithm [47]. This approach was implemented and tested with the result that it is much more successful than other random fuzzers [47]. Nevertheless, it still shows some weak points which can be also found at other black box tools like ensuring a specific percentage of coverage.

Another approach can be found in the paper "Grammar-based Whitebox Fuzzing" where the authors extend the white box fuzzing technique to make testing of applications with highly-structured inputs like compilers possible. The goal is to get deeper into the application, meaning to overcome the early stages. To generate test inputs, the white box fuzzing approach is extended. A grammar is needed which depicts valid inputs so that they are accepted in the language. Furthermore, tokens are marked as symbolic which are associated with variables. This leads to the possibility to comprehend the influence of tokens on control paths. The grammatical acceptance brings according to the authors [46] the following advantages:

- Reduction of the search tree based on invalid inputs
- Transforming satisfiable token constraints into valid inputs immediately

For testing purposes the JavaScript interpreter of the Internet Explorer 7 was used with the following outcome that grammar-based white box fuzzing has a higher degree of code coverage with less tests than black box fuzzing [46].

The presented approaches expand the basic types of fuzzing tools. Depending on the access rights of the tester, they can be used to overcome some difficult parts of testing, like code coverage.

3.1.2 Fields of Application of Fuzzing

The fuzzing technique can be used in different areas of computer sciences to successfully detect vulnerabilities. Good results have been achieved when using this approach for testing standard software, but it is also applicable for examining complex programs like CPU (Central Processing Unit) emulators or malware. For such a usage, various tools are needed which support additional functions to be suitable for more difficult areas. Tools need to be able to overcome checksum points or similar security measurements without causing errors to be able to search for vulnerabilities near the core function, deep within the software. Especially CPU emulators are hard to test as it is because of the complexity of a CPU not possible to test every single function.

An example of a fuzzer used for CPU emulators is EmuFuzzer. EmuFuzzer can handle process emulators and whole-system emulators. The fuzzer is used to prove that CPU emulators represent the CPU correctly. As it is not possible to test every scenario, the goal of the fuzzer is to show that the emulator does not satisfactorily emulate the CPU. The EmuFuzzer exhibit the restriction that at the moment it can only be executed in user-space completely. This means that the correctness of the emulation of unprivileged instructions can be proved, but for privileged instructions only the correctness of the prohibition can be verified [48].

Random and CPU-assisted test case generation were adopted to create test cases whereas in both cases the data is generated randomly. The definition of test cases is different in this paper than normally as test cases are not tests with a certain input but states instead during the test execution instead [48]. The algorithm used for generating test cases runs on both CPUs and works as follows [48]:

- All sequences are checked and those which do not stand for valid code will be thrown away
- The CPU act as oracle, it decides if such a sequence represents a valid instruction whereby those which evoke illegal instruction exceptions are not valid
- In the end, the results are compared whereby those cases where one CPU accepted the sequence and the other not are of special interest.

Testing of EmuFuzzer showed that it is possible to find defects with it but there are some limitations like instructions with pre-fixes are not supported. Therefore, further development is needed.

Another tool is FuzzASP which is a fuzzer for testing answer set solvers. The reason for its development is simple: The production of lots of different answer set programmes is necessary for grammar-based fuzzing in the field of answer set solvers according to the authors of "Testing and debugging techniques for answer set solver development" [43]. The authors point out that FuzzASP is able to create lots of different types of arbitrary program instances on the basis of huge program classes which makes it possible to offer an enormous amount of rule construct combinations. Those combinations can be equipped with choice, cardinality and weight atoms or the rules can be negated [43]. FuzzASP works as follows: Normal rules with empty bodies are created by taking each head from a set of normal atoms randomly. Furthermore, normal rules with non-empty bodies and different lengths are also established randomly. Then, each body atom and each atom is default negated with probability. Further steps depend on whether establishing a weight constraint program or a disjunctive program. Besides the fact that FuzzASP is configurable, it is also implemented in a way so that the generated programs are neither easy nor difficult to solve. To guarantee various severities, parameters are randomly chosen by the fuzzer whereby a minimum and maximum value for each one exist as explained by Robert Brummayer and Matti Järvisalo [43]. FuzzASP was tested with normal programs (NLP), weight constraint programs (WCP) and disjunctive programs (DLP), using the platform Ubuntu Linux [43]. The tests were successful finding defects like aborts and invalid answer sets.

Fuzzers can also be used to test malware. BitFuzz is a tool implemented for x86 binaries that tries to find errors in bad software. This is of special interest as normally only good software is covered in a research. With the used approach, BitFuzz is able to find vulnerabilities deep within the code of the program. This is possible because it is able to overcome basic integrity checks on the surface as described in "Input generation via decomposition and re-stitching: finding bugs in Malware" [49]. The approach is based on stitched dynamic symbolic execution which is used to make input generation possible if encoding functions are given. Firstly, encoding functions are recognized via trace-based dependency analysis during execution of a program. Afterwards, the investigation is augmented with the help of dynamic symbolic execution. Therefore, decomposition and re-stitching [49] is appended which works as follows:

- Decomposing is used to get the constraints which are needed for encoding, the others are handed on to a solver
- Re-stitching is used to generate new inputs consisting of a combination of the constraints and the values of the concrete execution of the encoded parts and their inverses

BitFuzz was tested on four different malwares which use complex encoding functions with the result that the approach was able to find bugs.

Fuzzing tools can also be adapted to reach insecure or possible attack points deeper in the software by getting over checksum points or encryption methods. TaintScope, for example, is an automated directed fuzzing tool which deals with checksums. The goal is to take checksums into account which do not fulfill a protection duty to prevent attacks on purpose but those checksums which should defend against accidental errors [41].

The goal is to overcome checksum-based integrity checks and the tool uses symbolic and concrete execution to be able to repair checksum fields in produced test cases. TaintScope consists of four phases. Firstly, an execution monitor is used for dynamic taint tracing, where it is recorded how programs deal with input data. Afterwards, a checksum detector is implemented to find checksum check points. Now, the phase "direct fuzzing" starts. There, the program gets contorted data as input which was created by a fuzzer module. This is only done if no bypass rules are established by the checksum detector. If rules exist, the fuzzer instead deliver the data to an instrumented program and changes the execution traces on the basis of the rules [41]. For generating test cases hot bytes information is modified that are those input bytes which contaminate function arguments, as described in "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection" [41]. Finally, the replayer is used to repair crashed samples meaning the tool fixes checksum fields which are the only value bytes seen as symbolic values. TaintScope was tested on different famous applications like Microsoft Paint and turned out to be effective in finding vulnerabilities.

Summarized, a huge amount of fuzzing tools for different types of software exists. The presented tools are specialized in various areas and seem to be quite useful. Nevertheless, they will not be taken into account for the evaluation in this thesis as most of them are not available for publicity. Besides, comprehensive testing of the fuzzing tools did not take place. The list of the tools shows that the fuzzing technique can be used for testing complex domains. As it is quite powerful, the fuzzing strategy is ideally suited for the field of security testing. Thus, for the practical evaluation in Chapter 5 security tools which work with the fuzzing approach were chosen to obtain advantages in revealing security vulnerabilities.

3.2 Presentation of Security Testing Tools

No matter if manual or automated testing is chosen, a wide variety of tools to ease and support the work of the tester exist. In this chapter, four fuzzing tools will be presented which shall be evaluated in a later step according to criteria which will be specified in Chapter 4. The tools were chosen by the author based on the condition that they shall be able to test web services. Furthermore, it must be tools which are classified as fuzzing tools. The testing tools have to be already implemented and tested, meaning not only an approach shall exist.

3.2.1 Fuzzolution

The fuzzer framework which was developed at Vienna University of Technology is a generic framework which supports different software paradigm like web applications and Voice over Internet Protocol(VoIP) via plugins. It pursues the goal to create a tool which traces errors. Fuzzolution is able to detect security vulnerabilities where different attacks such as denial of service(DoS) and cross-site scripting could be executed. In Figure 3.1 Fuzzolution within a test environment and the according communication for VoIP testing is illustrated for example.

The developers tested Fuzzolution based on two different softphones, namely QuteCom which is realized in Python, C and C++ and Session Initiation Protocol(SIP) Communicator, written in Java. The results of the tests were very satisfying because a huge amount of security vulnerabilities were found. Therefore, Fuzzolution is a good example for the evaluation conducted in the next chapter.

3.2.2 JBroFuzz

JBroFuzz is a fuzz testing tool of Open Web Application Security Project (OWASP) based on Java. OWASP is a non-profit organization which offers a huge amount of projects focusing on security. OWASP explains the aim

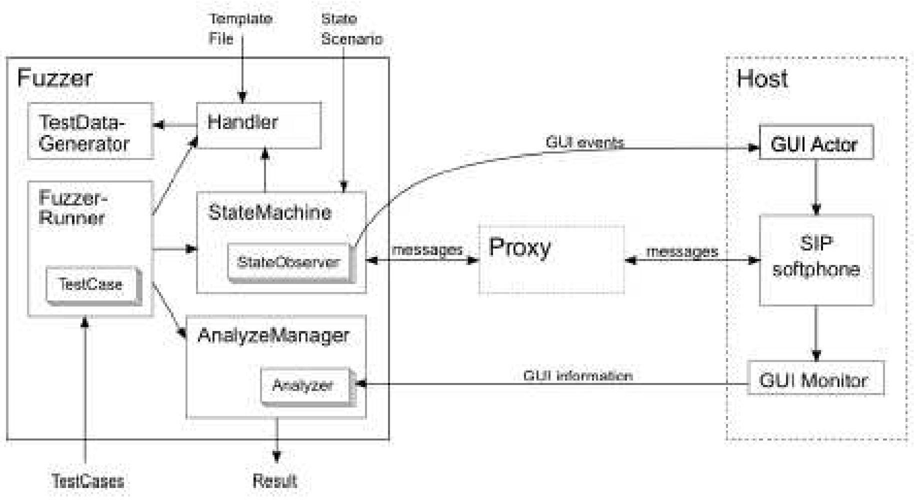
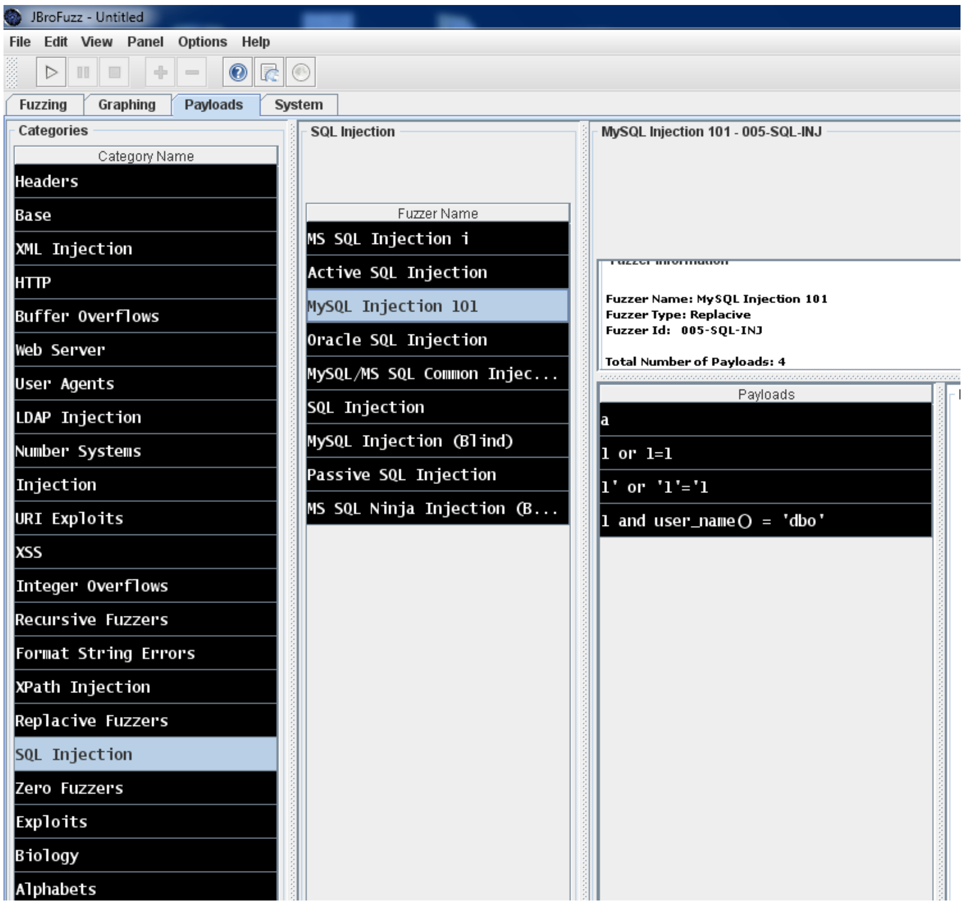


Figure 3.1: Fuzzolution within a test environment [50]

of this tool as establishing a fuzzer which is able to fuzz web protocols [51]. JBroFuzz is easy to learn and use. The only information needed to execute fuzzing is the Uniform Resource Locator(URL) and the request message. Until now, several releases exist whereas JBroFuzz 2.4 is the current one. According to the website [51] JBroFuzz executes requests and outputs the responses whereby the tester has to do further analysis to reveal possible weak spots. In the Frequently Asked Questions(FAQ) section [51] of the homepage of JBroFuzz readers can also find the hint that the tool does not do all the work for the tester, meaning basic background knowledge like definition of HyperText Transfer Protocol(HTTP) and HyperText Transfer Protocol Secure(HTTPS) is needed. JBroFuzz consists of five components, namely fuzzing, graphing, payloads, headers and system [51]. The fuzzing tab is the most important one as requests are established there. Additionally, the payloads are chosen on this tab according to the vulnerability which should be revealed. The diverse payloads are listed in the payloads tab as can be seen in Figure 3.2. They are sorted by popularly genera like SQL (Structured Query Language) Injection. By clicking on an attack type, further subtypes are at choice. The strings with which the application will be tested in the end are shown by selecting one of the subtypes.

3.2.3 Peach

Peach is a SmartFuzzer developed by Michael Eddington of Deja Vu Security. The current available version is Peach v2.3.8. On the website [52] Peach is described as a tool which masters both types of fuzzing, generation and


JBroFuzz - Untitled
File Edit View Panel Options Help
Fuzzing Graphing Payloads System
Categories
SQL Injection
MySQL Injection 101 - 005-SQL-INJ

Category Name
Headers
Base
XML Injection
HTTP
Buffer Overflows
Web Server
User Agents
LDAP Injection
Number Systems
Injection
URI Exploits
XSS
Integer Overflows
Recursive Fuzzers
Format String Errors
XPath Injection
Replacive Fuzzers
SQL Injection
Zero Fuzzers
Exploits
Biology
Alphabets

Fuzzer Name
MS SQL Injection i
Active SQL Injection
MySQL Injection 101
Oracle SQL Injection
MySQL/MS SQL Common Injec...
SQL Injection
MySQL Injection (Blind)
Passive SQL Injection
MS SQL Ninja Injection (B...

Fuzzer Name: MySQL Injection 101
Fuzzer Type: Replacive
Fuzzer Id: 005-SQL-INJ

Total Number of Payloads: 4

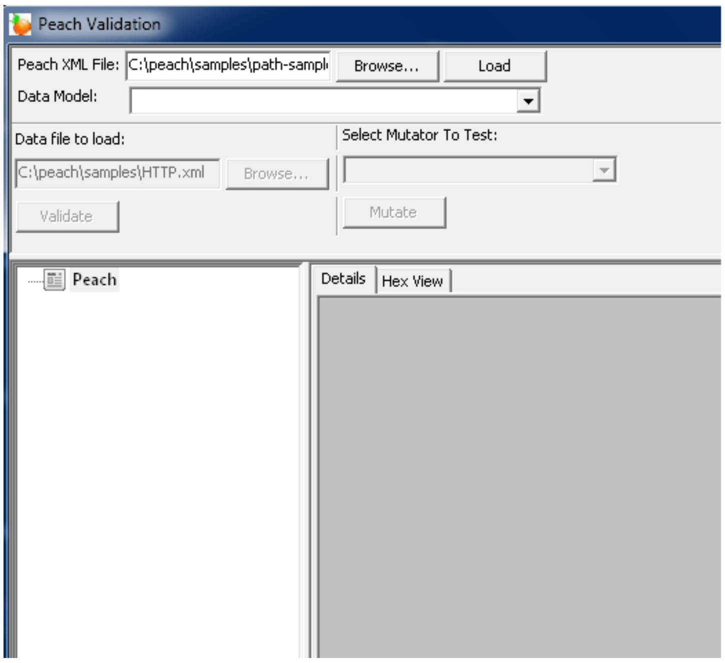
Payloads
a
1 or 1=1
1' or '1'='1
1 and user_name() = 'dbo'

Figure 3.2: Payloads of JBroFuzz [51]

mutation based fuzzing. Furthermore, the word ”SmartFuzzer” is explained as a fuzzer which has information about data types and their state for data which is changed during the fuzzing [52].

In the case of Peach, even more information is illustrated, like checksums and static transformations. Peach offers a good customer service as they provide developers who help users if they have problems with the tool or they assist them in developing or using Peach [52]. Additionally, trainings at conferences and onsite are offered to become familiar with the SmartFuzzer. To use Peach, XML (Extensible Markup Language) knowledge is required since the user needs to create Peach Pit files [52] to be able to start fuzzing. Peach Pit files are XML files which consists of several definitions, for example for data model, state model, agents and monitors. Furthermore, configuration details are added to the file. The tester can also set up a test case in this file. Mutators can be defined which shall be used during the execution of this test.

The Peach Pit files can be validated with the testing tool, either via command-line or via a graphical user interface as shown in Figure 3.3.



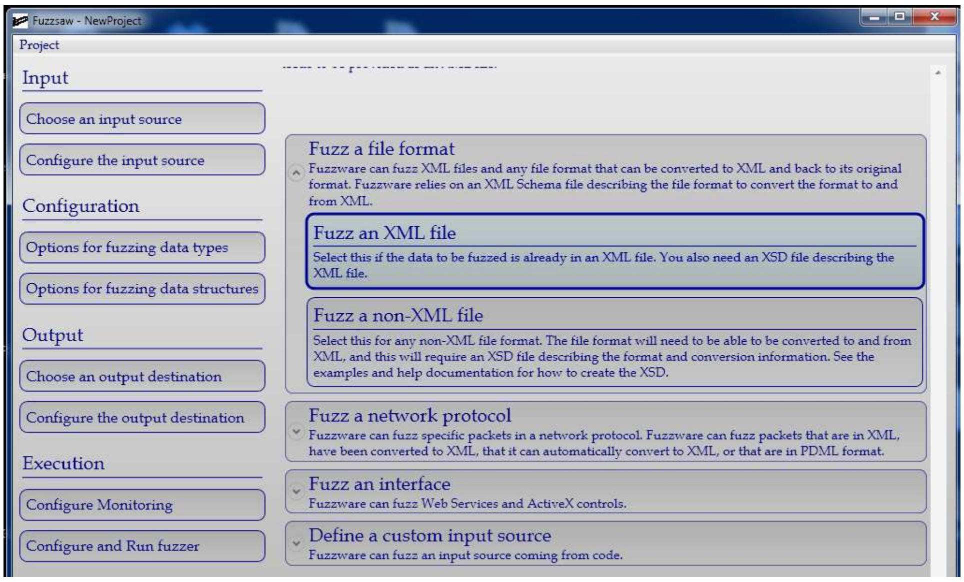
Peach XML File: C:\peach\samples\path-sampl
Data file to load: C:\peach\samples\HTTP.xml

Figure 3.3: Snippet of the validator of Peach [52]

The website of Peach [52] also states that in the beginning the tool used a top-down sequential method called SequencialMutationStrategy which leads to the fact that every element is considered at each test. Nowadays, also different mutation strategies are possible to use, whereas the standard by default is RandomDeterministicMutationStrategy, a mutator which fuzzes every element with all existing mutators. It is important to know that this technique is not the same as true random strategies as for each mutation only one single field is taken into account. As soon as every field has been fuzzed, the work of the fuzzer is done [52].

3.2.4 Fuzzware

Fuzzware is a generic fuzzing framework which specializes in web services whereby test cases are designed and executed automatically. The current version is 1.5. Fuzzware offers according to its website some specialities like the fact that it is able to convert different data formats which shall be considered for fuzzing into XML without human aid [53]. Additionally, data, data format and fuzzing information are not mixed in one single file but different files for each of the three points exist. Besides, no extra language was created to use Fuzzware. Instead it works with XML and XML Schema Definition(XSD) files. Lots of other features exist which are described at the website of Fuzzware [53] like the tool can be extended by the user, users can perform various actions on outputs, choose from 21 fuzzing types and it offers configurable fuzzing values. The tool includes event logs and a debugger. It is stateful and a possibility to examine configurations exists. The fact that fuzzware can deal with XML files but also transform non-XML files so that they can be used for fuzzing with this tool is of special interest as it makes the tool more flexible. The tool provides file fuzzing, packet fuzzing and interface fuzzing as can be seen in Figure 3.4. Furthermore, fuzzware is able to deal with custom .Net DLL inputs whereby DLL stands for Dynamic Link Library. As a prerequisite, it must implement Fuzzware.Extensible.IUserInputHandler. Fuzzware offers a lot of possibilities and features for testers who like to use fuzzing for ensuring quality of software.


Fuzzsaw - NewProject
Project
Input
Choose an input source
Configure the input source
Configuration
Options for fuzzing data types
Options for fuzzing data structures
Output
Choose an output destination
Configure the output destination
Execution
Configure Monitoring
Configure and Run fuzzer

Fuzz a file format
Fuzzware can fuzz XML files and any file format that can be converted to XML and back to its original format. Fuzzware relies on an XML Schema file describing the file format to convert the format to and from XML.

Fuzz an XML file
Select this if the data to be fuzzed is already in an XML file. You also need an XSD file describing the XML file.

Fuzz a non-XML file
Select this for any non-XML file format. The file format will need to be able to be converted to and from XML, and this will require an XSD file describing the format and conversion information. See the examples and help documentation for how to create the XSD.

Fuzz a network protocol
Fuzzware can fuzz specific packets in a network protocol. Fuzzware can fuzz packets that are in XML, have been converted to XML, that it can automatically convert to XML, or that are in PDML format.

Fuzz an interface
Fuzzware can fuzz Web Services and ActiveX controls.

Define a custom input source
Fuzzware can fuzz an input source coming from code.

Figure 3.4: Choosing what shall be fuzzed in Fuzzware [53]

Chapter 4

Definition of Evaluation Criteria

This chapter introduces criteria which are used for the evaluation process in the next chapter. The criteria are described together with information when they are considered to be fulfilled. Finally, priorities will be assigned to them. The requirements are chosen based on the goal to evaluate security testing tools which can be used for testing security aspects in XML-based web applications. This restriction does not allow to analyze every possible vulnerability because of the complexity of web sites as shown in Figure 4.1 whereby the first part indicates how simple the structure of web sites could be and the other one illustrates how complicated web sites are in reality [54].

The evaluation criteria presented in this thesis are a possible selection of requirements which could serve as evaluation basis. It is not a complete list of all usable criteria as then it would go beyond of the scope of this thesis. The criteria will be splitted in two groups as shown in Table 4.1 whereby the first group represents general criteria which can be used as selection basis for tools independent of the architecture paradigm. General criteria were selected by the author through searching of features which are absolutely needed for an automated testing tool to make it worth evaluating it independent of its use. The general criteria can be seen as pre-requisitions which must be fulfilled to shortlist and evaluate a tool. The other group will contain more specific requirements to ensure security, especially in web applications based on XML.

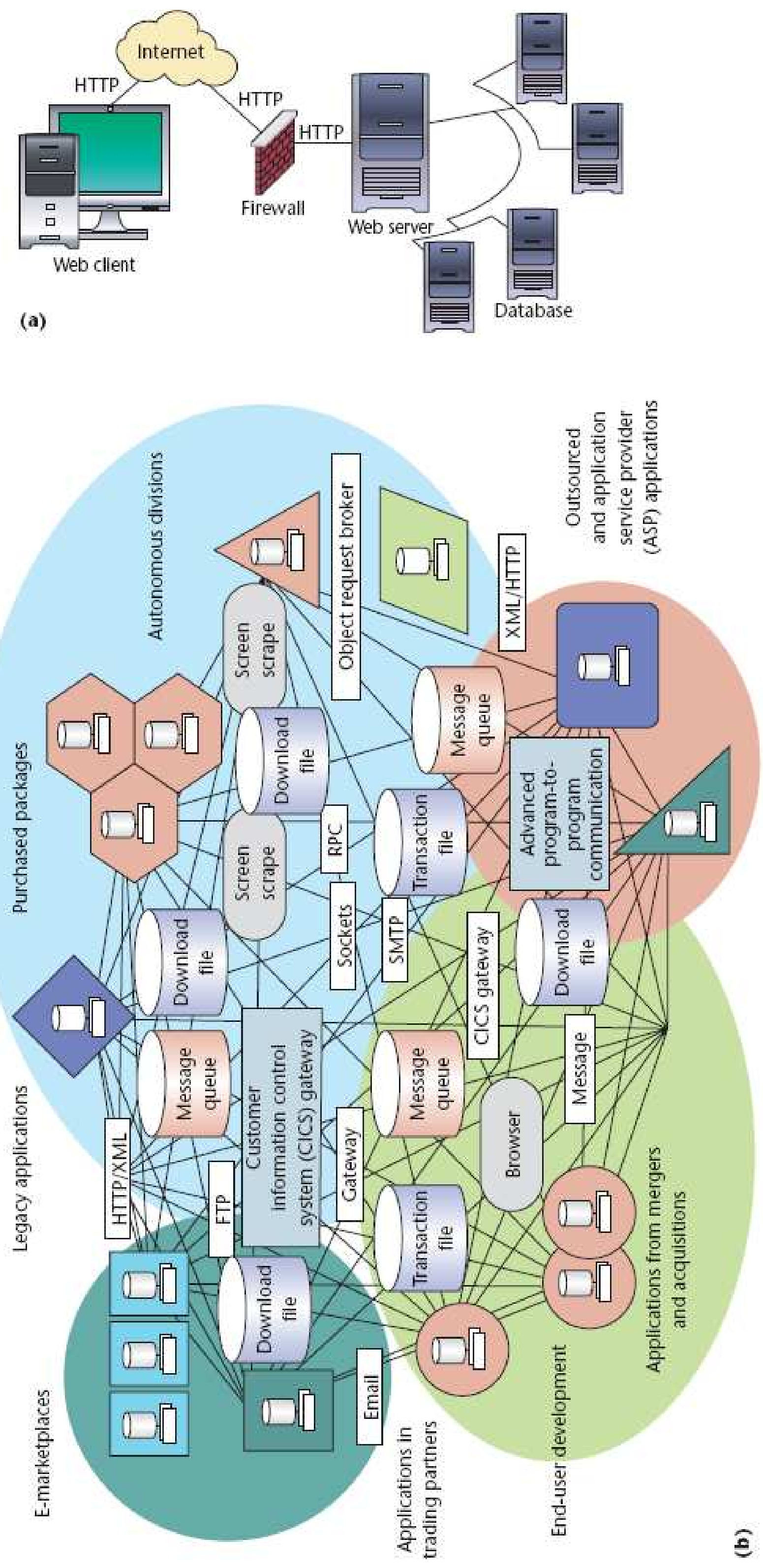


Figure 4.1: Desired and real web site structure [54]

General Criteria	**Vulnerability Detection Criteria**
Automation	XPath Injection
Usability	Support of XML generation based on schema
SUT Access Requirements	XML Manipulation
Platform	SQL Injection
Authentication	Cross-site Scripting
Interoperability	Denial of Service
Session Handling	Buffer Overflow
Presentation of Results	

Table 4.1: Classification of criteria

The criteria were selected based on common general web attacks which are described in different scientific papers [55] [56] [57], namely XPath injection, SQL injection, cross-site scripting, buffer overflow and denial of service attacks. In addition, some specific criteria which are useful for XML-based web applications, namely XML manipulation and support of XML generation based on schema were added. The general criteria, like usability, platform and interoperability were chosen from a scientific paper which handles test tool evaluation [58]. Furthermore, as the requirements are for automated test tools, automation, SUT access requirements and presentation of results were added by the author. Authentication and session handling are also taken into account as in relevant literature for web applications those criteria are often mentioned.

Together with a detailed description of all criteria, the way how it can be verified will be specified. This is necessary for the evaluation as a decision-making basis must be defined to be able to decide if the criterion is fulfilled. Nevertheless, the presented verifications are not the only possibility to check if a tool fulfils a criterion or not, there might be other ways for some of the criteria.

4.1 General Criteria

The following requirements which are also listed in Table 4.1 can be evaluated for every testing tool independent of the focused usage.

Automation Testing tools need to offer automation meaning that the execution of tests and also the comparison of test results must be done without the aid of the tester. During execution and result analysis, no manual input shall be necessary. In addition, the tool must save executed tests and results to offer the possibility to repeat tests and compare results. Automation of testing tools is especially interesting for tests where lots of transactions have to be done simultaneously like performance and stress tests as manual execution of such tests is not possible. This is also stated by Rudolf Ramler and Klaus Wolfmaier [59]. Additionally, the criterion can be extended to also include the need of an automated configuration so that no manual adaptions are needed. In this case, the configuration behaviour is irrelevant and will not be taken into account during the evaluation. The automation criterion is fulfilled when the tool executes tests and decides if the tests passed or failed without manual input.

Usability A tester has to have a certain degree of knowledge to use a testing tool depending on how easy or difficult it is conceived. This can lead to bad usability. Additionally, it might be quite time-consuming to install and use a complicated testing tool. Usability means also that the software works properly without errors and that users are satisfied with it. Basically, a testing tool shall ease and precipitate the testing process and thus no additional learning shall be needed. The term usability possesses a wider meaning which can be looked up in the general agreement from the standards boards American National Standards Institute(ANSI) 2001 [60] and International Standards Organisation(ISO) 9241 pt.11 [61] as also stated by Jeff Sauro and Erika Kindlund [62]. In the standards, the scope of usability as well as lots of criteria to measure it are described. As stated in "Usability Inspection Methods after 15 Years of Research and Practice" [63], diverse methods for testing usability exist like the following:

- Empirical usability testing
- Usability inspection methods

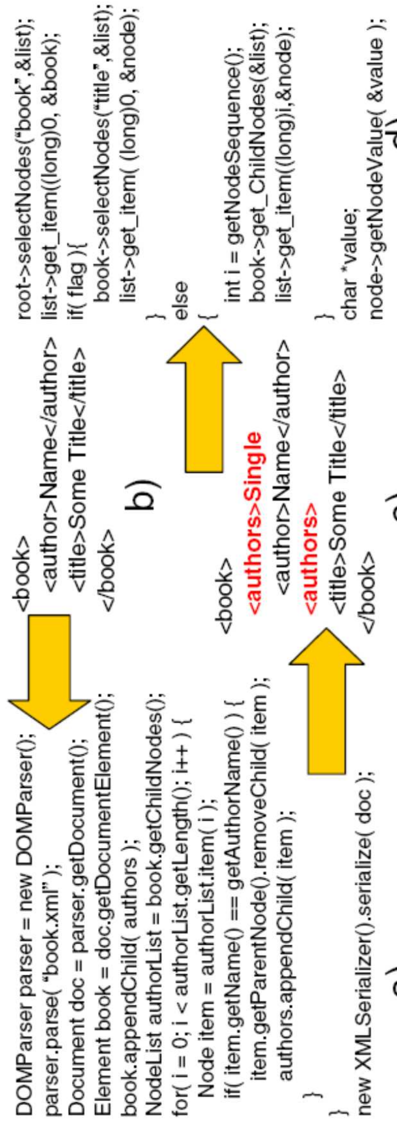
Which one to use is at the tester's discretion. Those techniques are beyond the scope of this thesis, so the criterion usability for evaluating testing tools will simply cover how easy or difficult the tool is to install and use, based on the impression of the author. As this is a subjective issue, usability is difficult to rate. Besides the experience of the tester also his or her personal preferences play an important role. Some might favour a tool which has a graphical user interface whereas others would appreciate a command line basis. It is also interesting in this case if a help page or something like that is available. Consequently, this criterion will only be described but not valued.

SUT Access Requirements The need of a certain level of access rights can be a prerequirement for the use of a tool. This depends also on the level of testing supported by the tool and required by the tester. Access rights can exist for different parts, for example for XML documents, databases or operating systems. Sometimes, the tester does not have access to the internal structure and code of the software, to the database or to the logfiles which might lead to a problem with a testing tool. Especially if the system under test is connected to a database, access to the database might be needed by the tool. The question is then, if read access is enough as for certain test cases it might be crucial to add or change data which would require write access. In some cases an exact copy of the data and structure of the database used in production might be useful to avoid an unwanted manipulation of production data by the tool. Additionally, the available degree of system access is important for the tester. For black box testing, it is not necessary for the tool to have access to the source code. In contrast, this is crucial if the tool can and shall execute white box testing as therefore code access is indispensable. Even more access rights like a login on the host might be needed. The more access is available, the more information is taken into account for the vulnerability detection. Besides, full access on all levels has the advantage that monitoring in more detail is possible. In addition, false positives can be reduced. To test this point, it is important to know if the tool supports black box testing, white box testing or both. This will give information about the needed access requirements. Furthermore, it is interesting to know if the testing tool can be used for testing production software or if it can only be used in special test environments. Again, this criterion is hard to rate as supporting only black box testing might be enough depending on the program to test. In contrast, white box testing is not possible without corresponding access rights, impartial of the tool. Nevertheless, a good testing tool should offer both to give the tester the choice and also make the use of the tool possible for testers who do not have extended access rights. The criterion is fulfilled if black box testing is possible so that no special access rights are needed.

Platform For the tester it is important to know on which platform the testing tool can run as sometimes the possibility to work on different platforms is not given due to security policies of a company. Supplementary, it is interesting which additional requirements for the environment must be fulfilled, for example which Java version is needed. Current versions are most of the time supported, but it is convenient if also older ones can be used to run the testing tool as companies might not update to the latest versions immediately because of special measures which must be captured before to guarantee impeccable support. This is especially with new database versions the case. Which versions are needed to fulfil the criterion depend on the test environment which the tester can use. The platform criterion cannot be evaluated because it is an individual criterion. Therefore, within the evaluation, it will only be described which platforms are supported by the tool and which further environment requirements must be given.

Authentication Authentication is the process of verifying the other parties. Different authentication methods exist, like passwords, public key cryptography and digital signatures. Passwords are one of the most common authentication methods where the user verifies itself with a password. This entails a lot of security risks, for example it can be guessed. Public key cryptography is a method where a public and a private key are used for authentication. For digital signatures, the message is hashed and a private key is then used on the hash value. With the public key, the identity can be checked. This criterion is of special interest when evaluating testing tools as lots of tools are not able to handle basic authentication methods which make them useless. The tool fulfils the authentication criterion if it is able to overcome the login method of web applications.

Interoperability Interoperability must be guaranteed to make both, software and testing tools useful. Interoperability states if systems or components can communicate with each other. Mark Grechanik shows an example of how interoperability works on the basis of XML in his paper "Finding Errors in Interoperating Components" [64]. He explained it with the aid of Figure 4.2 whereby Figure 4.2a) shows the Java part, Figure 4.2d) the C++ part of components which communicate via XML data as shown in Figure 4.2b-c). The arrows represent the data flow. Such a scenario is often seen in open source and commercial software. To test interoperability of testing tools it is important to check if there is a function, for example a button, which tests interoperability with the test objects. The tool should show if communication with certain objects is possible or not. This must be clear within a short period of time as it makes no sense if the tool indicates it two days later when the tests have already started or are even finished. It would result in a loss of time and wrong test outcomes. The goal is not to be interoperable with all test objects but to show if it is possible or not to communicate with them. If such a function exists, the tool fulfils the criterion.



```
DOMParser parser = new DOMParser();
parser.parse( "book.xml" );
Document doc = parser.getDocument();
Element book = doc.getDocumentElement();
book.appendChild( authors );
NodeList authorList = book.getChildNodes();
for( i = 0; i < authorList.getLength(); i++ ) {
   Node item = authorList.item( i );
   if( item.getName() == getAuthorName() ) {
      item.getParentNode().removeChild( item );
      authors.appendChild( item );
   }
}
new XMLSerializer().serialize( doc );
```

```
<book>
   <author>Name</author>
   <title>Some Title</title>
</book>
```

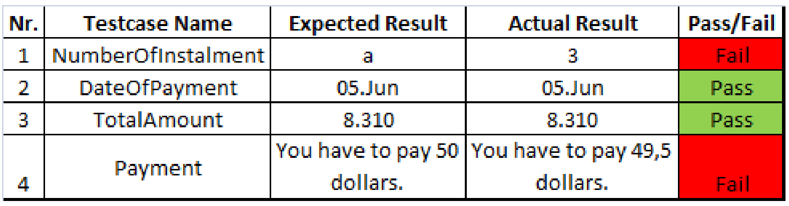
```
<book>
<authors>Single
   <author>Name</author>
<authors>
   <title>Some Title</title>
</book>
```

```
root->selectNodes("book",&list);
list->get_item((long)0, &book);
if( flag ){
   book->selectNodes("title",&list);
   list->get_item( (long)0, &node);
}
else
{
   int i = getNodeSequence();
   book->get_ChildNodes(&list);
   list->get_item((long)i,&node);
}
char *value;
node->getNodeValue( &value );
```

Figure 4.2: Example of Interoperability on the basis of XML data between different coding languages [64]

Session Handling Session handling is another crucial criterion to improve security. Proper session handling is important to prevent attacks, such as session hijacking, and resulting misuse of personal data. For correct session handling, a session ID is needed which is used to distinguish between users. Different possibilities how to transfer the session ID exist, like via the URL or via cookies. As normally several clients are using an application coincident, more than one unique session ID is needed. So if the tool is able to deal with different session IDs, the criterion is fulfilled.

Presentation of Results Another evaluation point is the presentation of results. It is desirable that test case results are displayed in a way so that it is immediately visible if the test passed or failed. Also the real output of the test might be shown. To search for various outcomes of the testing process exacerbates the work of the tester. For example, showing a list of test cases where successful tests are highlighted in green and failed ones in red would be a possible solution as illustrated in Figure 4.3.



Nr.	Testcase Name	Expected Result	Actual Result	Pass/Fail
1	NumberOfInstalment	a	3	Fail
2	DateOfPayment	05.Jun	05.Jun	Pass
3	TotalAmount	8.310	8.310	Pass
4	Payment	You have to pay 50 dollars.	You have to pay 49,5 dollars.	Fail

Figure 4.3: An example of how test results can be presented

Additionally, some kind of filter would be conceivable to select only failed test cases for instances and a possibility to change the ordering of the results. Another desirable attribute would be detailed information, meaning the tester can have additional information about the test case and its execution by, for example, clicking on it. An experienced tester will be able to work with results which are not highlighted in different colours and without additional features like the described ones but nevertheless, a good presentation of results might ease the work and even accelerate it. The presentation of results will only be described but not valued as every tester has a different preference regarding this point.

4.2 Vulnerability Detection Criteria

The following criteria are important for the evaluation of testing tools for XML-based web applications. It is necessary to determine which criteria a tool fulfils to be able to combine testing tools if there is no suitable one which meets every criterion. In this thesis, tools will be evaluated on the basis of the criteria listed in Table 4.1. For the evaluation, it is important to verify which criteria are implemented, meaning for example SQL injection is only important, if the application uses a database, proper session handling is crucial if sessions are used and XPath injection attacks can only be done if XML documents are browsed. The choice of evaluation criteria depends on the functionality which shall be tested.

XPath Injection An XPath injection attack, which could be compared to SQL injection attacks, is simple but dangerous, as the attacker has the possibility to get information out of all XML documents and hence can access personal data. XPath injection means that the attacker outwits the XPath expression by using malformed input. The resulting XPath query might be malicious in the sense of revealing information which should not be accessible for the attacker. For an XPath injection attack, strings like "or 1 = 1 or" or "//*" can be used. Such attacks can be prevented easily by validating every user input. Listing 4.1 shows an example of XPath injection and Listing 4.2 represents the appropriate code snippet where an XPath injection attack would not be possible. XPath is not only used for searching data in XML documents, but also for XQuery which is an XML language used for XML documents and databases to retrieve and alter data [65]. To satisfy this requisition, a tool must be able to create test cases with such insecure inputs or at least give the possibility to run such tests.

```
String e = "/users[@name='" + name + "'␣and␣" +
"@password='" + password + "']";
factory.newXPath().evaluate(e, doc);

    XPath in Java: XPath injection
```

Listing 4.1: Bad example of embedding XPath in Java [66]

```
XPath e = {- /users[@name=${name} and
@password=${password}] -};
factory.newXPath().evaluate(e.toString(), doc);

    XPath in Java
```

Listing 4.2: Good example of embedding XPath in Java [66]

Support of XML generation based on schema Every XML item must be validated against the chosen schema to guarantee that the code can be used. The grammar is especially important because if it does not match the rules, the XML document is useless. The validation of XML which was generated based on schema is also significant to prevent XML injection. Furthermore, valid XML is a good starting point to make interoperability possible, although components might not react on invalid XML. Whereas a parser would recognise XML which is not based on schema, interoperating components may not. The reason is that they do not always have to use the same data elements as explained by Mark Grechanik [64]. Besides, validation of XML items does not always guarantee interoperability as also described by Mark Grechanik [67] on the basis of Figure 4.4. If Application Programming Interface(API) calls are used for changing XML data, the data is not known early enough to use schema validators for proving the correctness of the data. Therefore, if the data does not correspond to the schema, a runtime error will occur in component J or in component C when runtime validation is executed or when API calls are executed. Regardless of the point of time, an error will occur for sure in this example independent of the schema validation.

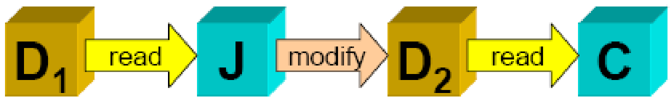


Figure 4.4: Interaction between components [67]

This shows that although the support of XML generation based on schema is important, it does not protect from every possible error occurring on the basis of invalid XML. A testing tool needs to be able to create XML based on a schema given by the tester to fulfil this criterion.

XML Manipulation Important for this criterion is the meaning of well-formed XML. In the book "Beginning XML" [68] the authors list the requirements respectively the rules which must be met to generate well-formed XML documents. The following six rules exist:

- Start-tags must have appropriate end-tags
- Tags are not able to interleave
- There is only one root element allowed per XML document
- XML naming conventions must be kept in mind for every element name
- XML distinguishes between the use of capital and small initial letters

- White space is not replaced by XML

Most of those rules are clear, like the fact that every start-tag must have an end-tag. Furthermore, it is important to close firstly the child elements and afterwards the parent elements, meaning the order must be kept. There must be always a root element, no matter if a content is given or not, for example the XML code in Listing 4.3 taken from the book "Beginning XML" [68]

```
<name>John</name>
<name>Jane</name>
```

Listing 4.3: Invalid XML code without root element [68]

must be changed to contain a root element like shown in Listing 4.4 [68] so that it is valid.

```
<names>
<name>John</name>
<name>Jane</name>
</names>
```

Listing 4.4: Valid XML code with root element [68]

To create names of elements, again some rules must be followed as described in "Beginning XML" [68], like names cannot start with numbers or other special characters than _, they are not allowed to start with "XML" in any combination and colon should be avoided.

Nevertheless, there are no reserved words which cannot be used. Case-sensitivity is especially important as otherwise the XML parser is not able to match the start-tags and end-tags properly. It makes a difference if the tag is spelled `<Names>, <NAMES> or <names>`. In addition, white spaces are not ignored in XML, no matter how much are consecutively.

One possibility of manipulating XML is XML injection which means that an attacker tries to attack the program by using malformed XML documents. The result can be both, well-formed and not well-formed XML. M. Jensen, N. Gruschka, R. Herkenhner and N. Luttenberger [69] explain XML injection as an attack where the attacker makes use of not properly defined tags, meaning the characters "<" and ">" are not correctly set. This enables the attacker to plant content and in further consequence to change the structure of an XML document which is then considered as part of the document. An example of insecure code can be found in Listing 4.5, a secure one in Listing 4.6. In the first Figure, an XML injection is executed which leads to a server error. This means that users cannot see the desired information. To avoid such problems, attacks must be prevented and not only detected as shown in Listing 4.6 where the attack is not successful at all [66]. Easy ways to prevent the success of XML injection attacks exist, like proper schema and data type validation [69].

```
String topic = getParam("topic");
String query = "SELECT body FROM comments " +
"WHERE topic = '" + topic + "'";
ResultSet results = executeQuery(query);
foreach (String body : results)
println("<tr><td>" + body + "</td></tr>");

XML and SQL in Java: XSS vulnerability
```

Listing 4.5: Bad example of XML and SQL embedded in Java Code [66]

```
String topic = getParam("topic");
SQL query = <| SELECT body FROM comments
WHERE topic = ${topic} |>;
ResultSet results = executeQuery(query.toString());
foreach (String body : results)
println("<tr><td>${body}</td></tr>".toString());

XML and SQL in Java
```

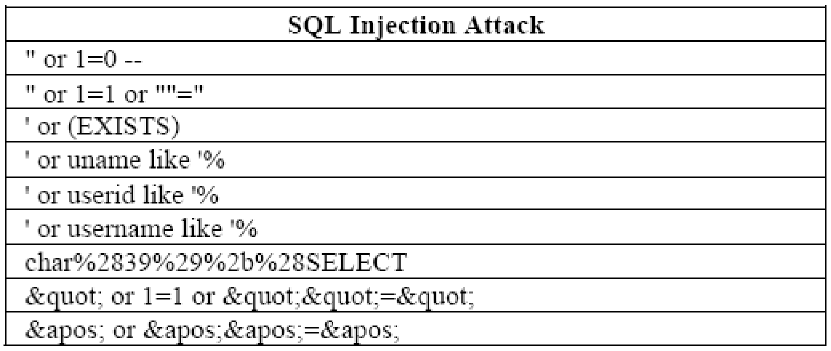
Listing 4.6: Good example of XML and SQL embedded in Java Code [66]

So XML Manipulation covers different possibilities like XML injection which can result in well-formed code, manipulation of XML which can result in not well-formed code or in general creating invalid messages. For this evaluation, the tool must either be able to generate invalid XML or to execute XML injection attacks to fulfil this criterion.

SQL Injection Many web applications need SQL to provide information from and save details to the database. Unfortunately, it is possible to manipulate the queries so that data is revealed which should not be public. This can be done by inserting certain strings such like the ones shown in Figure 4.5. Especially through input fields, for example, it is possible to outwit authentication methods or manipulate data, as also stated in "Trustworthy Web Services Based on Testing" [70]. The authors pointed out that SQL injection can be used to get data if user input is not validated properly when used in SQL queries.

Blind SQL injection is basically the same as SQL injection with the only difference that it is more difficult for an attacker to exploit such a vulnerability as the server does not answer with an error message containing useful information. Nevertheless, experienced attackers are able to execute such attacks.

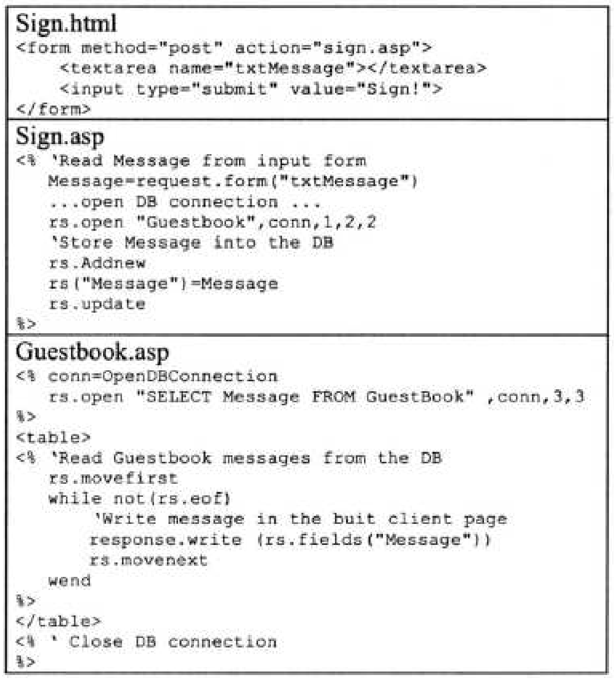
If a tool can create test cases which contain SQL injection attacks, meaning offer inputs which manipulate the queries or at least give the possibility to run such tests, then it meets the requirement.



SQL Injection Attack
" or 1=0 --
" or 1=1 or ""="
' or (EXISTS)
' or uname like '%
' or userid like '%
' or username like '%
char%2839%29%2b%28SELECT
&quot; or 1=1 or &quot;&quot;=&quot;
&apos; or &apos;&apos;=&apos;

Figure 4.5: Examples of possible input which can lead to SQL injection attacks [57]

Cross-site Scripting Cross-site scripting (XSS) is a very popular security vulnerability. For an XSS attack, the attacker tries to manipulate the application by putting malicious data in input fields so that this information is used on the client's side without changes. If the XSS attack is successful, malicious code can be executed on the side of the client. This can only be prevented if data is checked properly or encoded and the application does not trust every information it gets. Three different kinds of XSS attacks exist, namely stored XSS attacks, reflected XSS attacks and Document Object Model(DOM) based XSS attacks. If the data which is used for statements that generate Hyper-Text Markup Language(HTML) is not properly checked, a potential attacker has the possibility to execute malicious code on the client's side [71]. Reasons why cross-site scripting attacks are widely spread are given in "Static detection of cross-site scripting vulnerabilities" [72] like the fact that it is enough if the web application shows untrusty input without validation. Another reason is that programming languages hand on such input to the user without executing checks by default. So it is shown to the user exactly like it was given to the web application. Figure 4.6 shows a generic implementation of a guestbook in Active Server Pages(ASP). A possible XSS attack to the guestbook is presented in "Identifying Cross Site Scripting Vulnerabilities in web applications" [73] whereby only the string like shown in Figure 4.7 is needed to get the information of user cookies.



Sign.html

```
<form method="post" action="sign.asp">
    <textarea name="txtMessage"></textarea>
    <input type="submit" value="Sign!">
</form>
```

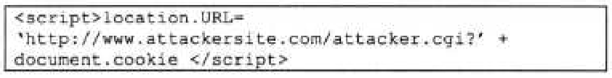
Sign.asp

```
<% 'Read Message from input form
   Message=request.form("txtMessage")
   ...open DB connection ...
   rs.open "Guestbook",conn,1,2,2
   'Store Message into the DB
   rs.Addnew
   rs("Message")=Message
   rs.update
%>
```

Guestbook.asp

```
<% conn=OpenDBConnection
   rs.open "SELECT Message FROM GuestBook" ,conn,3,3
%>
<table>
<% 'Read Guestbook messages from the DB
   rs.movefirst
   while not(rs.eof)
       'Write message in the buit client page
       response.write (rs.fields("Message"))
       rs.movenext
   wend
%>
</table>
<% ' Close DB connection
%>
```

Figure 4.6: Example of a web application [73]



```
<script>location.URL=
'http://www.attackersite.com/attacker.cgi?' +
document.cookie </script>
```

Figure 4.7: Example of a cross-site scripting attack [73]

In general, cross-site scripting is not difficult to correct but might get problematic in certain situations as stated in "Regular expressions considered harmful in client-side XSS filters" [74]. The authors compare the degree of severity of fixing XSS vulnerabilities to fixing buffer overflows. Nevertheless, the bigger a web site is, the more complicated it gets. Lots of web sites are not able to realise the fixing task completely. Additionally, repositories exist which list unpatched cross-site scripting vulnerabilities including the sites. This makes it even easier for attackers.

If a tool can generate test cases that could execute an XSS attack or at least give the possibility to run such tests, then it accomplishes this aspect.

Denial of Service Denial of service(DoS) is an attack where the raider tries to stop the service of the system which among other things harms the reputation of the provider. Various ways to reach this goal exist, for example creating an XML bomb with the aid of nested entities or the attacker simply takes away the key from a package which belongs to someone else and as a result is in the position to see the content of the document as explained in "Secure and selective dissemination of XML documents [75].

There are different kinds of denial of service attacks, like XML denial of service (XDoS) and distributed denial of service (DDoS) attacks.

- XML Denial of Service

  XML denial of service(XDoS) attacks are based on XML. The attacker is using XML to create lots of messages which consume than the parser completely until it cannot handle the load anymore. XDoS attacks are also explained in "Vulnerabilities Leading to Denial of Services Attacks in Grid Computing Systems: a Survey" [76] as attacks where the attacker sends a lot of requests to overcharge the parser and as a result the client cannot use the service anymore. The supported complex nested representation can be exploited by the attacker by increasing the nesting level so that the parser is not capable of handling the payloads and in further consequence crashes. According to the authors [76] it is also possible to send an enormous amount of messages.

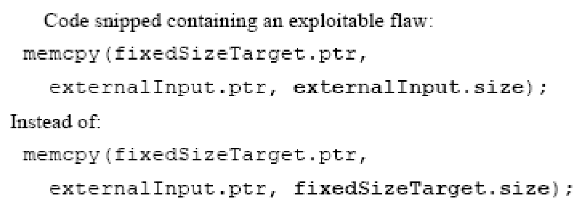
- Distributed Denial of Service

  For a distributed denial of service(DDoS) attack, the attacker is flooding the application not only from one machine but from various machines at the same time. This is where the term distributed comes from. In "A Practical Method to Counteract Denial of Service Attacks" [77] DDoS is described as an attack where the attacker makes use of several hosts which are often zombies. Zombies are compromised computers which exhibit defects. Most of them are not secured properly. With the aid of zombies, attacks can be executed on the client's machine.

A further aspect is the robustness of software programs which is wedded to DoS. Robustness is an important criterion to measure systems, especially large ones which deal with secure data, as it shows how much load they can handle and how reliable they are. M. Schillo, H.-J. Bürckert, K. Fischer and M. Klusch explained robustness as a criterion which can be evaluated by defining a certain benchmark, for example five percent random drop-outs, and then comparing the performance before and after it. With such explicit statements, robustness can be qualitatively measured [78]. If the software is not robust, it can quickly lead to a denial of service.

Error Handling is another interesting issue regarding denial of service attacks. This specific point is important for all software applications, no matter what they deal with. Correct error handling is necessary to classify the software as usable. Every single user makes mistakes, for example putting a number into an input field where an alphabetic character is required. In this case, the reaction of the software is important, as it should be able to deal with errors without crashing. Another reason why this is especially interesting for web services is given in "Experimenting with Exception Propagation Mechanisms in Service-Oriented Architecture" [79], namely the fact that web services might not be able to execute a rollback which leads to a need of proper exception handling as this is a way to make fault tolerance possible. For fulfilling this criterion, the testing tool has to be able to send XML messages which overcharge the parser.

Buffer Overflow A buffer overflow is a common vulnerability which occurs when the amount of data is huger than the available place in a buffer. In "Detection and prevention of stack buffer overflow attacks" [80] buffer overflows are defined as vulnerabilities which are caused by putting too much data into a fixed-size buffer. The memory next to the buffer is then overwritten and this might influence the program behaviour. For buffer overflows, a stack must be utilized during program execution. The dangerousness of this security issue has been reasoned by F.-H. Hsu, F. Guo and T. Chiueh [81] with the argument that lots of worms use buffer overflow attacks to broadcast. More general is the explanation of this attack in "Exploiting a buffer overflow using metasploit framework" [82] where buffer overflows are mentioned as one of the most widespread attacks. Buffer overflows also serve as basis for lots of other attacks. For a buffer overflow attack, the attacker gets remote access by detecting and using weak points of systems which are not coded properly. With the remote access, the attacker is in the position to abuse the system. Mustapha Refai [82] also stated that the attacker tries to get root privilege to do diverse harmful actions or even ruin the scheme. An example of a vulnerability is illustrated in Figure 4.8. In the first part, an external string is copied to a fixed-size buffer whereby the size was specified by the length of the external input. This is dangerous as a useable buffer overflow may result if the input is longer than the buffer. It would have been much more secure to use the size of the internal buffer as restriction as shown at the lower part in the Figure [83].



```
  Code snipped containing an exploitable flaw:
 memcpy(fixedSizeTarget.ptr,
   externalInput.ptr, externalInput.size);
Instead of:
 memcpy(fixedSizeTarget.ptr,
   externalInput.ptr, fixedSizeTarget.size);
```

Figure 4.8: Example of buffer overflow and how to prevent it [83]

Different types of buffer overflow exist, described in "Scalable Network-based Buffer Overflow Attack Detection" [81], namely code-injection (CI) attacks and return to libc (RTL) attacks. For code-injection attacks, the attacker uses the address space of the application by including corrupt code. The application's control is afterwards routed to this code. For return to libc attacks, the control is immediately navigated to a function which does already exist in the address space.

Buffer overflow is a very common method to acquire a denial of service. This attack can be easily executed which makes it even more dangerous. The buffer overflow criterion is fulfilled when the tool is able to generate huge input data.

4.3 Prioritization of Criteria

Depending on the target group, not every criterion is of high priority. The thesis focuses on users who are adept in testing. Therefore usability is not that important. Furthermore, the subject of testing are XML-based web applications where all the described attacks could happen, meaning the application is using all technical issues, like databases, sessions and XPath. The prioritization in Table 4.2 is just an example of how the criteria could be valued if the test object would be XML-based web applications. The prioritization is needed if more than one testing tool fulfils the criteria to have a decision basis whereby priority 1 stands for high priority (essential), 2 stands for medium priority (would be required) and 3 stands for low priority (nice to have but not absolutely needed). Usability is 3 as it does not influence the test results and does most likely not deter an experienced user from using a tool, if other significant requirements are met. Criteria like database and access rights get a 2 as they are decisive for the usability of the tool, but they can be influenced by the tester through requesting additional access rights for example. Automation is rated with 1 as the tool makes no sense if it cannot execute test cases without manual input. Criteria which directly affect the quality and results of the testing process, like SQL injection or denial of service get priority 1.

It is clear that a testing tool which fulfils more criteria with high priorities

Criterion	**Priority**
Automation	1
Usability	3
SUT Access Requirements	2
Platform	2
Authentication	1
Interoperability	1
Session Handling	1
Presentation of Results	3
XPath Injection	1
Support of XML generation based on schema	1
XML Manipulation	1
SQL Injection	1
Cross-site Scripting	1
Denial of Service	1
Buffer Overflow	1

Table 4.2: Prioritization of criteria

is more useful than others, for example a tool which meets all priority 1 criteria and some of the priority 2 requirements is more suitable than one which complies all priority 2 and priority 3 criteria. This categorization is just an example of how the criteria can be weighted on the supposition that the tester is experienced. It must be adapted according to the needs and experience of the person doing the testing and the application that shall be tested.

Chapter 5

Results of Security Testing Tools Evaluation

In Chapter five the author will accomplish an evaluation, record the results and determine if the testing tools meet each criterion. This exemplary evaluation shall show that the nominated criteria are useful and that they can be examined.

5.1 Evaluation Process

The tools will be evaluated by a combination of using the provided documentation and executing the tools. For this evaluation, all criteria described in the previous chapter will be taken into account. Afterwards, the documentation will be used to learn about which criteria are supported. Additionally, there will be a search on the Internet for publications and evaluations of the chosen tools to get information about which additional criteria are promoted. It is assumed that the information in the documentation and also from other sources mentioned is correct and therefore no additional tests for those criteria will be made. The whole evaluation process is illustrated in Figure 5.1.

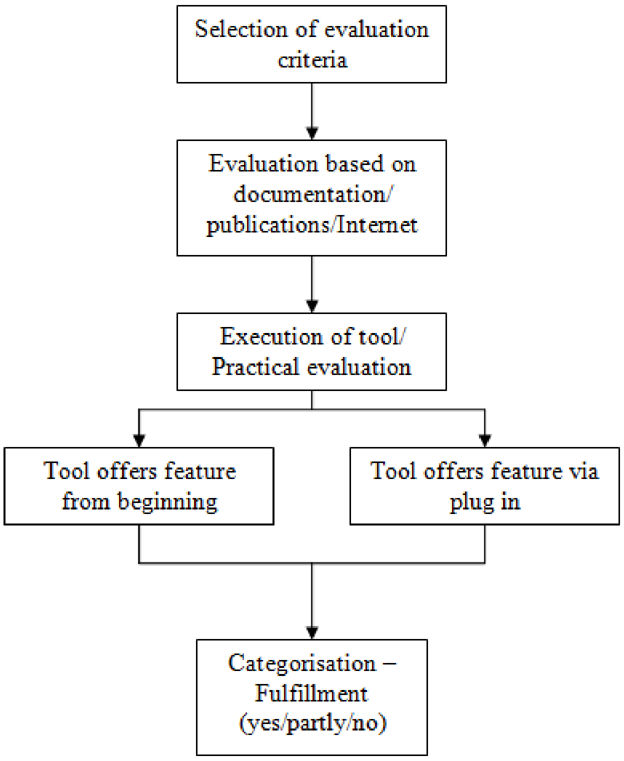

Selection of evaluation criteria
Evaluation based on documentation/ publications/Internet
Execution of tool/ Practical evaluation
Tool offers feature from beginning
Tool offers feature via plug in
Categorisation – Fulfillment (yes/partly/no)

Figure 5.1: Overview of the evaluation process

For SQL injection and blind SQL injection, a manual evaluation will be done to show how testing tools can be valued without documentation. This means, the tool will be executed and tested against the criteria. The evaluation itself is kept short as it should only serve as an example and is not the focus of this thesis. Some criteria are evaluated together in one step if it makes sense. Furthermore, tools will be categorised for each criterion whereby ”yes” stands for outstanding fulfilled, ”partly” stands for partial fulfilled and ”no” stands for not fulfilled/bad. Some of the criteria cannot be graded as the judgment is subjective, like usability. Nevertheless, in the comparison at the end, there will be subjective values for those criteria to make it clearer. The results of the evaluation of those criteria will be described in detail to show how the tools implement them. SQL injection will be evaluated practically to show how an evaluation can be done with a given list of needed requirements. Such an assessment shall be the basis for the tool decision. Additionally, it is interesting to know if the tool supports the criterion out of the box or if it can be added via plugins. For the author, a tool which can support a criterion with little additional effort, like changing the configuration, fulfills the requirement completely. If, instead, a huge workaround or even building a complex own fuzzer is necessary, the tool only partly fulfills the criterion.

5.2 Evaluation of Tools

This section shows the detailed results of the evaluated fuzz testing tools described in Section 3.2.

5.2.1 Fuzzolution

Fuzzolution is a tool which provides several plugins like VoIP or web service fuzzing.

Automation The tool is semi-automated and therefore partly fulfils the automation criterion. Several points are configurable but in general, generation of data, execution of the test cases, monitoring and analysis are automated.

Usability There is no graphical user interface, the tool is based on text files. Additionally, no syntax checks are done. The tool fulfils this criterion only partly. An advanced user might prefer the command-line based approach as used here as there are more possibilities to work with the tool, for example it can be easily included in scripts.

SUT Access Requirements This criterion is completely fulfilled. There are different variants, starting from black box approaches where less monitoring is done to white box attempts where logfiles and CPU workloads are monitored. As a result, the tool can be used by any tester no matter how much access to different levels like code and database is given.

Platform Fuzzolution is based on Java which means it can be used everywhere where Java is installed. Specific parts run only on Linux.

Authentication Authentication is out of the box supported by the tool for web applications. It can be extended for other applications, like Windows login. As web applications are the focus here, Fuzzolution fulfils the criterion.

Interoperability Fuzzolution is able to check if it is possible to communicate with the test object and gives feedback within a short period of time. As a result, the tool fulfils the criterion.

Session Handling The tool supports session handling and can hence be valued with yes regarding this point.

Presentation of Results There is no graphical support for the presentation of results which might make the tool a bit less intuitive. For experienced users it should be no problem to deal with the presentation of results of Fuzzolution. One possibility is to use text files. In the files, the evaluation information is logged, for example high CPU workload which results in the assumption that an error occurred. Nevertheless, if there is nothing stated in the logfiles, no error will be listed in this case. The second opportunity is to check the results in a database. There the user can see the test results in tabular form together with the probability that an error occurred. This is an issue which should not be categorised but as of missing graphical representation the tool partly fulfils the requirement.

Support of XML generation based on schema Fuzzolution supports the generation of XML items based on schemes or Data Definition Language(DDL). The XML items can be valid or invalid. Therefore, Fuzzolution completely fulfils this criterion.

XML Manipulation The tool can generate XML items which are not valid, meaning not well-formed. This can be on the basis of schemes or DDLs. Thus, Fuzzolution obtains a yes for this issue.

Cross-site Scripting Fuzzolution can execute XSS attacks via predefined attack lists or vectors. This leads to a complete fulfilment of the criterion.

XPath Injection All kinds of injections are supported by the tool via attack vectors. This functionality is given from the beginning and must only be configured depending if the tester wants them or not. Additionally, extensions through plugins or lists are possible. This means that a wide variety of attacks can be simulated and Fuzzolution fulfils the criterion.

Denial of Service Denial of service attacks are supported by Fuzzolution through XML bombs for example. The tester can then also check the CPU workload to realize such attacks. So memory usage is supported. The criterion is fulfilled.

Buffer Overflow It is possible to produce lots of huge and long strings via generators which can lead to buffer overflows. Therefore, the tool gets a yes for this criterion.

5.2.2 JBroFuzz

JBroFuzz is a fuzzer designed especially for web applications. The tool offers lots of possibilities for fuzzing.

Automation JBroFuzz is a semi-automated tool where the user can choose between different payloads which are later used for fuzzing. The results of the tests are written into files. Although, it is possible to see immediately if the request was accepted or not, the test results itself are not analysed by the tool. The tester has to check every single file and decide if a vulnerability exists or not.

Usability The version 2.4 offers both, command-line and graphical user interface. This means that the tool is comfortable and can be used by novice and experienced users as the graphical user interface is self-explanatory and visceral to use. Tutorials, FAQs and installation guides are available to help every tester to get started. Additionally, different tabs which provide various functions are available, like the "Graphing" tab. This would categorise the fuzzer as yes regarding usability.

SUT Access Requirements Basically, JBroFuzz is used for black box testing which means no special access rights are needed. Nevertheless, it is possible to write an own fuzzer and include JBroFuzz as library to also enable grey box and white box testing. This means the tool does not need any special access requirements to run properly, but it can be extended which would then require additional access rights. Therefore, the criterion is fulfilled.

Platform This criterion is fulfilled as the tool runs on different platforms, like Win 32, Mac OSX or Backtrack 3 [51]. Furthermore, Java 1.6 or greater is required.

Authentication Authentication is supported by JBroFuzz. In the tutorial section of the OWASP website [51], examples are described, explaining how fuzzing with this tool works regarding generic proxies. Some of those proxies need user authentication. Therefore, the authentication requirement is fulfilled as basic authentication is promoted. More complex ones are not in the scope of the tool at the moment and deductive the tool gets a partly.

Interoperability In general, it is possible to see if the tool is able to communicate with a test object as basic results of tests are immediately visible. With human analysis regarding the details of the result, the user can see if something went wrong with the communication, for example it can be seen that there was no communication because of a missing Internet connection. This leads to the fact that the criterion is fulfilled.

Session Handling When fuzzing with JBroFuzz, the desired URL can be denoted. As the URL can include the session ID to enable proper session handling, the tool fulfils the criterion.

Presentation of Results Within the graphical user interface, the results are represented in tabular form. By clicking on one of the entries, the user gets further details about the test and its result. Although no special colours are used for the representation, the tool gives a good and intuitive overview of results. Additionally, the user has the possibility to face a graphical representation of results in the appropriate tab. Nevertheless, the tool does not state if a vulnerability is given on a specific site or not. For such information, further human analysis of the results is needed. As JBroFuzz gives several opportunities to view test results which are also easy to understand for novices but still manual analysis is needed, the tool partly discharges this criterion in the opinion of the author.

XPath Injection In "OWASP JBroFuzz Tutorial", some examples of how to use JBroFuzz are explained step by step. Also the fact that XPath injection is supported can be found here [51]. Besides this, JBroFuzz can be used as fuzzing library. There are core fuzzing APIs which can be utilized to build own fuzzers. With this option, everything is possible and even if not out of the box supported, all injection attacks can be tested with JBroFuzz with a bit developing effort. Therefore, the tool gets a yes in this case.

Support of XML generation based on schema There is no support to generate XML based on schema, at least it is not mentioned on the OWASP Website for JBroFuzz [51]. Nevertheless, it is possible to add a new fuzzer which creates such XML. This is a lot of effort and as a result JBroFuzz gets only a partly for this criterion.

XML Manipulation The generation of not well-formed XML is not concretely mentioned but with an easy workaround it would be possible to fuzz with such XML items. Recursive fuzzers can be used to replace entries with different characters, alphabetical, numerical or special characters. This leads to not well-formed XML and therefore the tool accomplishes this point partly.

Cross-site Scripting Cross-site scripting is supported by the testing tool. It offers several different fuzzers, like replacive fuzzers, double fuzzers and power fuzzers, from which the tester can choose from. With them, cross-site scripting is possible as also stated in the "Payloads and Fuzzers" section of the Website [51]. Different variants of cross-site scripting attacks are delivered with the tool. Again, it is possible to create own fuzzers which support more complex attacks, therefore, the tool fulfils the criterion.

Denial of Service In JBroFuzz, no own payload for this criterion exist. Nevertheless, taking the payload "Long Strings of aaa's" which consists only of a's with a payload length of 65,537 should be sufficient to lead to a denial of service. Therefore, the criterion is fulfilled.

Buffer Overflow JBroFuzz supports buffer overflows. A variety of own payloads for this kind of attack exist. So buffer overflow attacks can be executed and the tool accomplishes the requirement completely.

5.2.3 Peach

Peach is a tool which comes with several features. It is extensible which makes it usable for advanced fuzzing [52]. This means also that the tool can fulfil a lot of criteria with creating own fuzzers but not out of the box.

Automation Peach is a semi-automated tool. It can generate data with mutators, provides a file system logger and offers monitoring for example to trace crashes or to gather network traffic. Furthermore, own monitors can be written by the tester [52]. Nevertheless, Peach does not offer automatic comparison of results meaning human analysis is needed which leads to the fact that the tool only partly fulfils the requirement.

Usability Peach is command-line based. With the next release, it is planned to provide a graphical user interface for at least simple file and network fuzzing. This tool is quite complex which makes it difficult to use especially for novices. With the current release, there is a graphical user interface for some parts available but still most of the functionality works only command-line based. Therefore, the tool accomplishes this point partly.

SUT Access Requirements There are no special prerequisites to use Peach. As the tester has to write Peach Pit Files at least a basic development environment is needed. Additionally, to write own fuzzers and XML files it might be necessary to have access to source code and database so that special information, like details about data model and structure can be received. As a result, the tool is evaluated with partly.

Platform Peach supports Windows, Unix and OS X [52]. Additionally, Python is required [52]. This means that Peach can run on a wide variety of platforms and therefore it completely fulfils the criterion.

Authentication Authentication is not supported out of box. Nevertheless, it is possible to build an own fuzzer which can overcome any authentication methods. An example of how the authentication part of the fuzzer could look like can also be found in the tutorial section of the Website of Peach [52]. The tool is valued with partly for this requirement.

Interoperability Peach is based on Pit Files which must be created by the user to use the tool. No button or similar function is available which checks interoperability with the test object but it is suggested to validate Pit Files and their data, no matter which type of fuzzer is created. The first mutator does by default not fuzz any data. This is used to check if interoperability between the fuzzer and the target is given [52]. So the user can with the given tool examine interoperability but this process is not automated. Instead, human analysis is needed which leads to partly for this criterion.

Session Handling With peach, it is possible to put the session ID into the URL which means the tool can deal with session handling. Therefore, the criterion is fulfilled.

Presentation of Results The results of test cases are saved in a file [52]. Additionally, logging is possible [52]. This means that there is the opportunity to monitor and check the results of test cases. However, the tester has to open every file to see the results, there is no possibility to immediately see if it passed or failed. In addition, there is no graphical representation which justifies for the author a not fulfilled for this criterion.

XPath Injection and Cross-site Scripting Peach does not support such attacks out of the box. As it is more a framework to build own fuzzers than an all inclusive fuzzer ready to use, the possibility exists to develop a fuzzer which is able to execute such attacks. Despite the tool offers different kinds of mutators like StringMutator or PathMutator which can be used to generate mutated data. This will help to realize such attacks with a small workaround and little development effort but to completely test XPath injection and cross-site scripting more realistic attacks must be depicted and hence the tester has to design a complete new fuzzer with own mutators. The tool gets a partly for those criteria as it is possible to generate fuzzers which satisfy the requirements for such attacks.

Support of XML generation based on schema The generation of XML based on schema is supported by Peach. The user has to build his or her own Peach Pit Files where the schema can be designed which shall be the basis for the creation of XML and fuzzing process. The tool is scored with yes here.

XML Manipulation The user needs to give a data model to Peach so that the tool is able to mutate data. Depending on how the data model is described, not well-formed XML can be created and used for test cases with Peach. As there is no function where the user only has to copy the data model and the tool does the rest, Peach is valued with partly in this case.

Denial of Service As Peach is a fuzzer framework, it should be possible to write an own fuzzer and own mutators or maybe even use given mutators which are able to execute denial of service attacks. The criterion is consequently partly fulfilled.

Buffer Overflow Again, a fuzzer which can fuzz for buffer overflows can be designed by the user as well as all other types of fuzzers. The criterion is partly accomplished as there is no mutator which is specialised in buffer overflows but using other existing mutators might be sufficient.

5.2.4 Fuzzware

Fuzzware can test files, network packets or calls to interfaces. It can be used to fuzz web services.

Automation Fuzzware is a framework which automatically creates and executes test cases. Again, the tester has to check the results, the tool does not automatically analyse them, but there is the possibility to check for certain words in the event log. This leads to partly fulfilled for this criterion.

Usability Fuzzware comes with a user interface for its configuration. In the user interface, the tester can choose the type which shall be used for fuzzing, for example file fuzzing or network fuzzing. Unfortunately, the user interface does not support XML document creation [53]. Nevertheless, over the user interface, the user can choose input and output source, configure the fuzzing, monitoring, testing and executing. Regarding usability, the tool fulfils the requirement.

SUT Access Requirements There was no information published about required access rights. Therefore, it is assumed that it is not a prerequisite to have special access rights when using the tool. Fuzzware gets a yes regarding SUT access requirements.

Platform Fuzzware is only for Windows available.

Authentication Authentication methods are supported by Fuzzware. The tool requires a Web Services Description Language(WSDL) file including among other things methods to fuzz web services. The tester can then state the methods which should be initially called to influence the order of the invoke of those methods [53]. As this is supported, the tool completely accomplishes the criterion.

Interoperability Fuzzware offers a test mode which can be switched on and off in the user interface. If the test mode is on, no fuzzing will be done. This is to check the configuration which shows also if communication between components work [53]. It is possible to define a file which shall be compared with the test mode result. Furthermore, the tool is able to convert non-XML data format to XML to make them fuzzable. The tool fulfils the criterion.

Session Handling Fuzzware can handle different session IDs via the URL. The tool gets a yes for this criterion.

Presentation of Results Different ways for presenting results of test cases exist. Firstly, a command prompt appears during execution as any output is monitored to stdout/stderr. With the current release, this output text is now also coloured. Additionally, a log file is available and another file which is named like the state of the test case containing the results whereby the user can configure the output directory and file extensions [53]. Fuzzware offers diverse possibilities for presenting testing results but to find out if the test case was successful or not, every file has to be opened which means there is no possibility to see the results immediately. Therefore the tool fulfils the criterion partly regarding the requirements of the author.

Support of XML generation based on schema This criterion is completely fulfilled by Fuzzware. For executing fuzzing, an XML and an XSD file is needed whereby the XML file comprises data and the XSD file is used for defining types and structure. Fuzzware can use the information of the XSD file for creating test cases [53]. As a result, the tool is valued with yes.

XML Manipulation The generation of not well-formed XML is a bit tricky in Fuzzware as the tool sticks to the schema. So it is better to not define the format too exactly as this enables more possibilities for fuzzing. Fuzzware suggests to define the elements more exactly than the types of them [53]. When defining the schema a bit different then it should be, a generation of not well-formed XML is possible. Fuzzware is valued with yes in this case.

XPath Injection Based on methods created by the user which include XPath expressions, the tool can offer support for such vulnerabilities. However, Fuzzware just mutates the input data according to the available fuzzing techniques so there might be no complicated XPath injection attacks if the user does not generate XPath expressions with certain logic behind it. So Fuzzware partly fulfils the criterion.

Cross-site Scripting Fuzzware offers no special function for XSS attacks. The user can only create them self. Nevertheless, it is at least possible to execute them. There is no out of the box support and in the end the user has to generate the attack by himself which leads to a partly fulfilled criterion in this case.

Denial of Service Denial of service attacks can be realized with Fuzzware and the tester has also the opportunity to add own values for fuzzing easily. This leads to fulfilled criterion in this case.

Buffer Overflow It is possible to evoke a buffer overflow through replacing certain data with long strings. Hence, Fuzzware completely accomplishes the buffer overflow requirement.

5.2.5 Evaluation of the Criterion SQL Injection and Blind SQL Injection

SQL injection and blind SQL injection will be tested with the help of an application called "Damn Vulnerable Web App (DVWA)" [84]. DVWA is based on PHP: Hypertext Preprocessor(PHP)/MySQL and has targets which correspond to the purpose of evaluating security testing tools. The application was created to help security professionals, developers, teachers and students to teach, understand and test security knowledge and tools [84]. The current version is v1.0.7. DVWA offers several security leaks like XSS, SQL injection and file inclusion. A web server, PHP and MySQL is needed to run DVWA properly. The most convenient way is to use XAMPP as it combines all in one. The user can choose if the vulnerability is easy, medium or difficult to reveal. Additionally, it is possible to view the source code and a help page for every weakness.

Test Environment

For the evaluation of the tools, SQL injection and blind SQL injection with low security level was used. The goal is to find a way to get username and password of five users which are identified by numerical IDs from "1" to "5" in the database. A possible solution is to use the string "' or 'a'='a" for SQL injection and the string "1' or '1' = '1" for blind SQL injection to get all values out of the database. The tests were done on a Windows XP Professional machine. To use DVWA, a login is needed. As login methods are not part of the tests, DVWA was manipulated by switching off the login directly in the code. Furthermore, the difficulty level was set to low permanently in the source code to ease the test process.

JBroFuzz

For evaluating JBroFuzz the current version 2.4 was used. After starting the test tool, the target must be put in which is in this case "http://localhost". Then, the request field has to be filled as shown in Listing 5.1.

```
GET /dvwa/vulnerabilities/sqli_blind
/?id=z&Submit=a&level=0
HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.0;
en-GB;
rv:1.9.0.10) Gecko/2009042316 Firefox/3.0.10
(.NET CLR 3.5.30729)
/2.5
Accept: text/html,application/xhtml+xml,application
/xml;q=0.9,
*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Listing 5.1: Request in JBroFuzz

It is not necessary to do this manually, there is also the possibility to click on File/Open Location and put into the URL field the URL, for example `"http://localhost/dvwa/vulnerabilities/sqli_blind/?id=z&Submit =a&level=0 HTTP/1.1"` for blind SQL injection attacks. By clicking on "OK" the fields will be set automatically. Now, the value which should be fuzzed has to be marked which is the "z" for this test. By right-clicking on it and choosing "Add" the desired fuzzers can be added. The author chose MS SQL Injection i, MySQL Injection 101 and MySQL Injection (Blind) from the category SQL Injection. For every fuzzer, the encoder URL UTF-8 is set because otherwise the website could not be called. Finally, the fuzzing can start by clicking on Panel/Start. In the "Output" tab, an overview of the used payloads and appropriate results can be seen. Unfortunately, there is only the status code shown which means the tester knows then if the request got to the website but not if the attack shows any results. To figure this out, every single entry must be right-clicked and opened in the browser. If a user wants to fuzz with a huge amount of values, this can be a very time-consuming and exhausting work. So the tool is only semi-automated as the interpretation of the test results has to be done manually. JBroFuzz was able to conduct successful attacks for the SQL injection and the blind SQL injection section of DVWA. Therefore the tool fulfils the criterion completely.

Peach

For evaluating Peach version v2.3.8 was installed into the test environment. The example HTTP.xml [52] served as basis for the test. Firstly, the Peach Pit file has to be adapted which will serve as a fuzzer in the end. A data model has to be defined as illustrated in Listing 5.2. The code shows snippets of the example for the blind SQL injection attack.

```
<!--Create a simple data template containing a
single string-->
<DataModel name="HttpRequest">

<!-- The HTTP request line: GET http://foo.com
HTTP/1.0 -->
<Block name="RequestLine">

<!-- Defaults can be optionally specified via the
value attribute -->
<String name="Method" mutable="false"/>
<String value="GET http://localhost/dvwa/
vulnerabilities/
sqli_blind/?id="
type="char" mutable="false"/>
<String name="RequestUri" mutable="false"/>
<String value=" " />
<String name="HttpVersion" mutable="false"/>
<String value="HTTP/1.1\r\n" mutable="false" />
</Block>

<!-- This block uses the Header block as a base
and overrides one field -->
<Block name="HeaderHost" ref="Header">
<String name="Header"
value="http://localhost" isStatic="true"/>
</Block>
```

Listing 5.2: Example of data model definition for Peach

Furthermore, a publisher has to be defined which determines how data is sent and retrieved as can be seen in Listing 5.3.

```
<Publisher class="tcp.Tcp">
    <Param name="host" value="127.0.0.1" />
    <Param name="port" value="80" />
```

```
</Publisher >
```

Listing 5.3: Example of publisher for Peach

The tester can decide where test results shall be put whereby it can be chosen between various opportunities like putting results into a file or to stdout. Unfortunately, the output does not really give much information besides the used mutator and the input values, if set up. In the error.log file of the web server much more information can be found, like the used URL with the fuzzed values and the HTTP status code which at least let the user know if the request was successful or not and which error occurred. Peach itself does not offer an error.log file. The author changed the low.php file of DVWA so that the output of the request is written into an .html file. This eases the review of the test results.

Additionally, a test has to be set up where also the mutators are defined as shown in Listign 5.4.

```
            <!-- Create a simple test to run -->
<Test name="HttpGetRequestTest"
 description="HTTP Request GET Test">
  <StateModel ref="State1"/>

  <Mutator class="string.StringMutator" />

  <Publisher class="process.DebuggerLauncher"
  name="launch"/>

</Test>
```

Listing 5.4: Example of a test case for Peach

As the goal is to fuzz the value of the id which is of the data type string, the StringMutator was used. There are even more details which can be defined in the Peach Pit files, the presented one shall just give an overview of the most interesting ones. After all needed values are set, the file can be validated to see if parsing errors exists. This is done through the command defined in Listing 5.5.

```
C:\peach>peach -t HTTP.xml
```

Listing 5.5: Command for validating Peach Pit files

If the file is OK, the fuzzing can start. If only one iteration is needed than the command illustrated in Listing 5.6 can be used. It is also possible to give a range as argument so that several test numbers are done as shown in Listing 5.7.

```
C:\peach>peach -1 HTTP.xml
```

Listing 5.6: Command for Peach to run one interation

```
range:C:\peach>peach --range 1 8 HTTP.xml
```

Listing 5.7: Command for Peach to run several tests

With the chosen StringMutator, the results were not satisfying. There was no list where values can be chosen which shall be taken into account for fuzzing. As only the predefined strings are used, neither the SQL injection attack nor the blind SQL injection attack was successful in revealing all database entries. This is because the mutator does not offer complex string inputs. The StringMutator would however be successful if a buffer overflow attack would have been the goal as it mutates the given values with long strings. This does not mean that Peach cannot be used for SQL injection attacks but it is necessary for such attacks to write an own mutator with appropriate string values and combinations for fuzzing the id. For this, the user needs knowledge of Phyton. Besides, more effort than for the other presented tools is needed to get Peach running as it requires more detailed information. Therefore, the tool gets only a partly here.

Although Peach is a powerful framework, it is more suitable for attack types which shall be tested in detail and recommended for experienced users only as much more time and effort is needed to get familiar with it than for other presented tools. The tool is semi-automated as the fuzzing is done automatically but test results have to be analysed manually.

Fuzzware

For testing Fuzzware version 1.5 was used. The first step was to adapt the example file HTTPPostParams.xml [53] as this example was used as a basis for the test. In the header part the URL was set as shown in Listing 5.8 for blind SQL injection attacks and as illustrated in Listing 5.9 for SQL injection attacks.

```
/dvwa/vulnerabilities/sqli_blind/ HTTP/1.1
```

Listing 5.8: Definiton of URL for blind SQL injection in Fuzzware

```
/dvwa/vulnerabilities/sqli/ HTTP/1.1
```

Listing 5.9: Definiton of URL for SQL injection in Fuzzware

The HeaderField looked than as depicted in Listing 5.10.

```
<http:HTTPHeader>
 <http:HeaderField>
    <![CDATA[POST /dvwa/vulnerabilities/sqli_blind/
   HTTP/1.1]]>
 </http:HeaderField>
```

Listing 5.10: Definition of HeaderField for blind SQL injection test

Furthermore, the parameter "id" which shall be fuzzed in a later step was set. By changing "dontFuzz" from true to false, the tool is told to fuzz the id and not run only one test case with the hardcoded values as illustrated in Listing 5.11.

```
<?Schemer Id="Body"?>
  <http:HTTPBody>
    <?Schemer dontFuzz="false"?>
    <pp:PostParams>
      <pp:ParamName>id</pp:ParamName>
      <pp:Equals>=</pp:Equals>
      <pp:ParamValue>1</pp:ParamValue>
    </pp:PostParams>
  </http:HTTPBody>
```

Listing 5.11: Setting the ID in Fuzzware

In the corresponding configuration.xml file, the input files are declared as shown in Listing 5.12.

```
<XMLFileInput>
<XMLPathAndFilename>HTTPPostParams.xml</XMLPathAndFilename>
<XSDPathAndFilename>..\HTTP.xsd</XSDPathAndFilename>
<XSDPathAndFilename>PostParams.xsd</XSDPathAndFilename>
</XMLFileInput>
```

Listing 5.12: Definition of input files for Fuzzware

Now the fuzzing can start. As soon as a new project is set up, the point "Fuzz an XML file" is chosen. Under "Configure and Run fuzzer" the Test Mode must be set to off to enable fuzzing. By clicking on "Options for fuzzing data types" a screen is shown where the data types can be chosen which shall be fuzzed. The tester can choose between Strings, Integers, Decimals and Bytes type fuzzers whereby combinations of some of them or using all fuzzers is possible. Additionally, own values for fuzzing can be added or even a whole set of them by clicking on the button "Add custom fuzzing values". In this case, only the string values were taken into account, all other data types were set to off. The reason was that only string values are interesting for this specific test and using all fuzzers would need a long time to execute and bring no advantage here. During the fuzzing, all given String values were adapted, meaning not only the values itself for the id or the host but also the name "id" for example. The replacements were combined in different orders whereby not always all strings were substituted for every test case. The tests were positive as both, SQL injection attacks and blind SQL injection attacks were successful. All database entries were shown. The test results were written into an XML file. Therefore, an own programme was written by the author to change those files to .html files which makes the visualisation of the results more readable. The name of the files shows the tester which test was executed, for example the name of the file "pp-PostParams-0-Occurrence-2.txt" means that all params were combined two times consecutively. The file "OutputWithoutFuzzing.txt" shows the results for the hardcoded values. Unfortunately, a check of each file was needed to find out which input gives the desired output. Once the functionality of Fuzzware is clear, it is quite easy to use and successful in testing SQL injection and blind SQL injection attacks which leads to a fulfilled requirement.

Fuzzolution

Fuzzolution possesses predefined attack vectors. Those can be used for SQL injection and blind SQL injection. Furthermore, it supports HTTP requests whereby it was possible to execute the tests. The tool is a framework and can be extended easily. In this case, the needed components for executing the tests have already been in place. For the generation of the test cases, attack vectors were used. The tests showed that the tool is automatized as it detects errors without manual input. This is realised through different analyser implementations. To find out if the tests were successful, the response of the server is analysed. The tool checks if there are discrepancies between the response for valid requests and the responses it gets from test cases. The tests were successful as both, the SQL injection and blind SQL injection vulnerability were found by Fuzzolution.

Nevertheless, the quality of the error detection depends on the configuration. As no support for the configuration is offered by the tool, it can easily come to wrong configuration which leads to the fact that the tool is not successful anymore. Another important point is the duration of the test execution. In this case, the test cases were executed fast as only a few attack vectors were used and one field of the HTTP request was specifically tested. Normally, several fields need to be fuzzed with a lot of different variants to execute proper security tests. This influences the duration and can lead to an execution time of some hours or even days. The duration depends also on the possible speed of requests through the SUT.

All in all, Fuzzolution is able to detect SQL injections and therefore completely fulfils the criterion. Although it has some disadvantages or rather inconveniences, the tool is suited for executing security tests.

5.3 Comparison of Results

In general, it is not easy to compare tools just on the basis of documentation because for the chosen tools the documentation was incomplete and there was nearly no information about them on the Internet. This leads to the fact that for a detailed evaluation, every tool must be executed and tested on the basis of every single criterion. A time-consuming task, as it is not enough to execute the tools with simple examples. Instead there must be complex ones which are bound up with huge efforts to guarantee that the tool satisfies the requirements of the tester. Depending on the amount of criteria, this can be an endless process. On the other side, tools with good documentation exist which would then make a comparison much easier and faster.

The tools evaluated have all advantages and disadvantages. The suitability of them for testing certain architecture paradigms depends heavily on the knowledge of the user and the attacks which shall be executed. All in all, Fuzzolution shows the best results for this evaluation. The fact that it is highly configurable makes it even more attractive for testers. Nevertheless, the tester must have some experience in the field of fuzzing to use it appropriately. Peach got the worst results which can be also traced to the fact that it is more a framework for building own fuzzers than a ready-to-use fuzzer. This tool might be most interesting for advanced and experienced testers as it is quite powerful. It is possible to use it for complex testing but due to difficulty level of its usage compared to the other tools, it will take much more time in the sense of prework to get excellent results with it. JBroFuzz was especially designed for web applications and is easy to use. It has the advantage, that no further knowledge is needed to execute some low level tests quickly. Therefore, it is a good security testing tool for novices. On the other side, it does not fulfil all of the chosen criteria which concludes that it is not so powerful in revealing deeply hidden security vulnerabilities. Fuzzware can also be used to test web applications but it is limited to Windows machines and needs more development from tester's side. In contrast, it is easy to add own fuzzing values for this tool. This fact is very important as the allocated values are often not enough.

Criterion	**Fuzzolution**	**JBroFuzz**	**Peach**	**Fuzzware**
Automation	partly	partly	partly	partly
Usability	partly	yes	partly	yes
SUT Access Requirements	yes	yes	partly	yes
Platform	Java	Java	Python	Windows
Authentication	yes	partly	partly	yes
Interoperability	yes	yes	partly	yes
Session Handling	yes	yes	yes	yes
Presentation of Results	partly	partly	no	partly
XPath Injection	yes	yes	partly	partly
Support of XML generation based on schema	yes	partly	yes	yes
XML Manipulation	yes	partly	partly	yes
SQL Injection	yes	yes	partly	yes
Cross-site Scripting	yes	yes	partly	partly
Denial of Service	yes	yes	partly	yes
Buffer Overflow	yes	yes	partly	yes

Table 5.1: Comparison of evaluation results

In Table 5.1 an overview of the evaluation results can be found. It shows which tool fulfils which criteria completely, partly or not at all to ease the decision of the tester. This presentation is not mandatory but it gives a good summary to quickly decide for a security testing tool. Alternatively the prioritization of each criterion could also be added to an extra column. Although Fuzzolution seems to be the clear winner at a first glance, this is only partly true. From the perspective of a novice who wants to execute some basic security tests quickly JBroFuzz might be more appropriate. Experts, in contrast, will most likely prone to Peach. Furthermore the importance of every single requirement must be taken into account. According to the prioritization done in the previous chapter, the decision for Fuzzolution is clear. With other priorities, this fact might change. In addition, the author wanted a tool which can be used for XML-based web applications and is able to detect a long list of security attacks. Again, with different criteria other evaluation results might have been achieved. This shows also that the evaluation process is adaptable so that it can be used by every tester regardless of the purpose. In the end, the final decision which security testing tool to use must be taken by the tester depending on knowledge, experience, time and willingness to learn. The evaluation results can only lead as an assistance for the choice by giving an overview of criteria fulfilled by the tools. In some cases it might be more efficient to combine some tools to get the desired results of testing.

The evaluation process is successful as it is possible to evaluate security testing tools with it. The fact that the documentation for the chosen tools was not useful for all criteria is not taken into account here as it is not necessary that every requirement is explicitly mentioned. Experienced testers will quickly find out if a criterion is supported or not by using their knowledge. In addition, with a detailed research, it is possible to get the necessary information. Furthermore, enough tools with better documentation exist. Of course, the convenience of using a tool cannot be evaluated with the process for the simple reasons that this is individual, strongly dependent on the preference of the tester, and to get a feel for any security testing tool it must be practically tested. Nevertheless, the goal to find out quickly which tool fulfils the selected criteria is reached by the presented evaluation process.

Chapter 6

Conclusio

Testing is essential for software development processes as one step in a process to improve software quality. Successful testing is only possible if a certain process is followed from project start until the end of the development process. Automated test tools exist to ease the work of the tester and make certain tests possible, like load tests even for huge programs.

Different test strategies exist, one of it is called fuzz testing where test data is invalid or unexpected and randomly chosen. An enormous amount of tools which concentrate on fuzzing are available. For this thesis four tools, namely Fuzzolution, JBroFuzz, Peach and Fuzzware were chosen and described whereby all of them have different strengths and weaknesses. All of them can be used to test XML-based web applications. The tools were evaluated on the basis of the presented evaluation process.

The evaluation process was kept easy to enable a quick and proper assessment as it is not the goal to lose plenty of time when choosing an appropriate tool. It works on the basis of evaluation criteria. Those requirements are used to make a decision. The suggested criteria are just examples, they must be extended or exchanged depending on the software type to test. Every criterion should have a priority to make a comparison of the tools possible.

The evaluation showed that all selected tools are efficient if XML-based web applications are tested. Nevertheless, none of the tools was able to fulfil all criteria. Especially the requirement "Automation" was only partly accomplished because no selected tool was in the position to analyse the results. Instead, further human analysis is needed. For this evaluation, Fuzzolution offered the best results as it fulfilled most of the chosen criteria completely. Peach satisfies most of the criteria only partly. Nevertheless, it is a useful tool as it is highly expandable but only for experienced testers who possess a huge developping knowledge. For novices, Peach is too complex and a long period of time is needed to be able to deal with it. The author prefers JBroFuzz as the way how it works is easy to understand and everybody can start fuzzing with it quickly without additional effort. Altogether, the final choice should be made by the tester depending on the experience and preferences of him or her.

The evaluation process turned out to be useful for testers. It functions as a guide for selecting the appropriate testing tool. Some of the tools needed further development to be able to find all the attacks described. This can lead to additional education effort. All in all, the choice of appropriate testing tools needs some time if a tool satisfying all needs shall be found. Besides, tools must be analysed in detail, meaning reading only the help pages is often not enough to make the right decision. They differ a lot especially in handling and offered features. In the end it is still the personal preference which will be the decisive factor for the decision. Nevertheless, for a quick decision the process can be a huge help, especially in cases where testing must start in a short period of time and no further development effort is desired.

Chapter 7

Terms and Abbreviations

- XML Extensible Markup Language
- SUT Software Under Test
- SQL Structured Query Language
- IEEE Institute of Electrical and Electronics Engineers
- POSIX(R) Portable Operating System Interface for Unix
- MTP Master Test Plan
- UAT User Acceptance Test
- CPU Central Processing Unit
- NLP Normal Programs
- WCP Weight Constraint Programs
- DLP Disjunctive Programs
- VoIP Voice over Internet Protocol
- SIP Session Initiation Protocol
- OWASP Open Web Application Security Project
- URL Uniform Resource Locator
- FAQ Frequently Asked Questions
- HTTP Hypertext Transfer Protocol

- HTTPS HyperText Transfer Protocol Secure
- XSD XML Schema Definition
- DLL Dynamic Link Library
- ANSI American National Standards Institute
- ISO International Standards Organisation
- API Application Programming Interface
- XSS Cross-site Scripting
- DOM Document Object Model
- HTML HyperText Markup Language
- ASP Active Server Pages
- DoS Denial of Service
- XDoS XML Denial of Service
- DDoS Distributed Denial of Service
- CI Attack Code-Injection Attack
- RTL Attack Return To Libc Attack
- DDL Data Definition Language
- DVWA Damn Vulnerable Web App
- WSDL Web Services Description Language
- PHP PHP: Hypertext Preprocessor

List of Tables

List of Figures

Listings

Bibliography

[1] E. L. Jones and Ch. L. Chatmon. A perspective on teaching software testing. In *Proceedings of the seventh annual consortium for computing in small colleges central plains conference on The journal of computing in small colleges*, pages 92–100, , USA, 2001. Consortium for Computing Sciences in Colleges.

[2] B. Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold, 1984.

[3] M. Fewster and D. Graham. *Software Test Automation - Effective Use of Test Execution Tools.* Addison-Wesley, 1999.

[4] J. A. Rosiene and C. Pe Rosiene. Testing in the 'small'. *J. Comput. Small Coll.*, 19(2):314–318, 2003.

[5] A. Bertolino. Software testing research and practice. In *ASM'03: Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, pages 1–21, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] H. H. Thompson. Why security testing is hard. *IEEE Security and Privacy*, 1(4):83–86, 2003.

[7] M. N. Kreeger. Security testing: mind the knowledge gap. *SIGCSE Bull.*, 41(2):99–102, 2009.

[8] J. Srinivasan and N. Leveson. Automated testing from specifications. *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, 1:6A2–1 – 6A2–8, 2002.

[9] E. Dustin. *Effective software testing: 50 specific ways to improve your testing.* Addison-Wesley, 2002.

[10] W. Lam. Testing e-commerce systems: A practical guide. *IT Professional*, 3(2):19–27, 2001.

[11] I. Londesbrough. A test process for all lifecycles. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 327–331, Washington, DC, USA, 2008. IEEE Computer Society.

[12] T. A. Majchrzak. Best practices for the organizational implementation of software testing. In *Proceedings of the 2010 43rd Hawaii International Conference on System Sciences*, HICSS '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[13] E. Miller. Workshop report: Software testing and test documentation. *Computer*, 12:98–107, 1979.

[14] B. Littlewood, editor. *Software reliability: achievement and assessment*. Blackwell Scientific Publications, Ltd., Oxford, UK, UK, 1987.

[15] P. R. Pfau. Applied quality assurance methodology. *SIGSOFT Softw. Eng. Notes*, 3:1–8, January 1978.

[16] M. Pyhäjärvi, K. Rautiainen, and J. Itkonen. Increasing understanding of the modern testing perspective in software product development projects. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 8 - Volume 8*, HICSS '03, pages 250.2–, Washington, DC, USA, 2003. IEEE Computer Society.

[17] M. W. Whalen, A. Rajan, M. P. E. Heimdahl, and S. P. Miller. Coverage metrics for requirements-based testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36, New York, NY, USA, 2006. ACM.

[18] M. R. Lyu. Reliability-oriented software engineering: Design, testing and evaluation techniques. *Software, IEE Proceedings -*, 145:191 – 197, 1998.

[19] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[20] S. Baharom and Z. Shukur. Module documentation based testing using grey-box approach. *International Symposium on Information Technology, 2008*, 2:1 – 6, 2008.

[21] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[22] S. Beydeda and V. Gruhn. Integrating white- and black-box techniques for class-level regression testing. *Computer Software and Applications Conference, 25th Annual International*, 0:357 – 362, 2001.

[23] N. H. Petschenik. Building awareness of system testing issues. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 182–188, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[24] IEEE Computer Society. Ieee standard for information technology - requirements and guidelines for test methods specifications and test method implementations for measuring conformance to posix(r) standards. 1998.

[25] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. *40th Annual Hawaii International Conference on System Sciences*, 0:261a, 2007.

[26] P. Joshi, K. Sen, and M. Shlimovich. Predictive testing: amplifying the effectiveness of software testing. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 561–564, New York, NY, USA, 2007. ACM.

[27] T. Xie. Improving effectiveness of automated software testing in the absence of specifications. *Software Maintenance, IEEE International Conference on*, 0:355–359, 2006.

[28] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 571–579, New York, NY, USA, 2005. ACM.

[29] C. Kaner. Pitfalls and strategies in automated testing. *Computer*, 30:114–116, 1997.

[30] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. *SIGSOFT Softw. Eng. Notes*, 27:97–106, July 2002.

[31] L. Borner, T. Illes-Seifert, and B. Paech. The testing process - a decision based approach. In *Proceedings of the International Conference on Software Engineering Advances*, pages 41–, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Futcher and R. von Solms. Guidelines for secure software development. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*, SAICSIT '08, pages 56–65, New York, NY, USA, 2008. ACM.

[33] G. Davis. Managing the test process. In *Proceedings of the International Conference on software Methods and Tools (SMT'00)*, pages 119–, Washington, DC, USA, 2000. IEEE Computer Society.

[34] A. Mette and J. Hass. Testing processes. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 321–327, Washington, DC, USA, 2008. IEEE Computer Society.

[35] IEEE Computer Society. 829-2008 ieee standard for software and system test documentation. 2008.

[36] K. R. P. H. Leung and W. L. Yeung. Generating user acceptance test plans from test cases. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 02*, COMPSAC '07, pages 737–742, Washington, DC, USA, 2007. IEEE Computer Society.

[37] J. Bergeron, H. Foster, A. Piziali, R. S. Mitra, C. Ahlschlager, and D. Stein. Building a verification test plan: trading brute force for finesse. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 805–806, New York, NY, USA, 2006. ACM.

[38] M. Smith and N. Thompson. The keystone to support a generic test process: Separating the "what" from the "how". In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 342–353, Washington, DC, USA, 2008. IEEE Computer Society.

[39] J. W. Cangussu. Modeling and controlling the software test process. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 787–788, Washington, DC, USA, 2001. IEEE Computer Society.

[40] Y. Jiang, Y. Li, S. Hou, and L. Zhang. Test-data generation for web services based on contract mutation. In *SSIRI '09: Proceedings of the 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, pages 281–286, Washington, DC, USA, 2009. IEEE Computer Society.

[41] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 497–512, Washington, DC, USA, 2010. IEEE Computer Society.

[42] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random*

testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pages 1–1, New York, NY, USA, 2007. ACM.

[43] R. Brummayer and M. Järvisalo. Testing and debugging techniques for answer set solver development. *Theory Pract. Log. Program.*, 10:741–758.

[44] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional, 2007.

[45] N. Rathaus and G. Evron. *Open Source Fuzzing Tools.* Syngress Publishing, 2007.

[46] P. Godefroid, A. Kiežun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 206–215, 2008.

[47] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. pages 477 – 486, 2007.

[48] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing cpu emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 261–272. ACM, 2009.

[49] J. Caballero, P. Poosankam, S. McCamant, D. Babić, and D. Song. Input generation via decomposition and re-stitching: finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 413–425. ACM, 2010.

[50] S. Taber, Ch. Schanes, C. Hlauschek, F. Fankhauser, and T. Grechenig. Automated security test approach for sip-based voip softphones. In *Proceedings of the 2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, pages 114–119. IEEE Computer Society, 2010.

[51] OWASP. Jbrofuzz. `http://www.owasp.org/index.php/JBroFuzz`. Last accessed on 2011-09-20.

[52] M. Eddington. Peach. `http://peachfuzzer.com/`. Last accessed on 2011-09-20.

[53] Fuzzware. `http://www.fuzzware.net/`. Last accessed on 2011-09-20.

[54] M. Curphey and R. Araujo. Web application security assessment tools. *IEEE Security and Privacy*, 4(4):32–41, 2006.

[55] S. Kals, E. Kirda, Ch. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 247–256, 2006.

[56] N. Kabbani, S. Tilley, and L. Pearson. Towards an evaluation framework for soa security testing tools. *2010 IEEE International Systems Conference*, pages 438 – 443, 2010.

[57] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira. Effective detection of sql/xpath injection vulnerabilities in web services. In *SCC '09: Proceedings of the 2009 IEEE International Conference on Services Computing*, pages 260–267, Washington, DC, USA, 2009. IEEE Computer Society.

[58] R. M. Poston and M. P. Sexton. Evaluating and selecting testing tools. *IEEE Softw.*, 9:33–42, May 1992.

[59] R. Ramler and K. Wolfmaier. Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 85–91, New York, NY, USA, 2006. ACM.

[60] ANSI (2001). Common industry format for usability test reports (ansi-ncits 354-2001). American National Standards Institute.

[61] ISO (1998). Ergonomic requirements for office work with visual display terminals (vdts) part 11: Guidance on usability (iso 9241-11:1998(e)).

[62] J. Sauro and E. Kindlund. A method to standardize usability metrics into a single score. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–409, New York, NY, USA, 2005. ACM.

[63] T. Hollingsed and D. G. Novick. Usability inspection methods after 15 years of research and practice. In *SIGDOC '07: Proceedings of the 25th annual ACM international conference on Design of communication*, pages 249–255, New York, NY, USA, 2007. ACM.

[64] M. Grechanik. Finding errors in interoperating components. In *IWICSS '07: Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques*, page 3, Washington, DC, USA, 2007. IEEE Computer Society.

[65] J. Robie. Xml processing and data integration with xquery. *IEEE Internet Computing*, 11:62–67, July 2007.

[66] M. Bravenboer, E. Dolstra, and E. Visser. Preventing injection attacks with syntax embeddings - a host and guest language independent approach. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 3–12, New York, NY, USA, 2007.

[67] M. Grechanik. Finding errors in components that exchange xml data. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 264–273, 2007.

[68] K. Cagle, N. Ozu, J. Pinnock, D. Gibbons, and D. Hunter. *Beginning XML*. Wrox Press Ltd., Birmingham, UK, UK, 2000.

[69] M. Jensen, N. Gruschka, R. Herkenhöner, and N. Luttenberger. Soa and web services: New technologies, new standards - new attacks. In *ECOWS '07: Proceedings of the Fifth European Conference on Web Services*, pages 35–44, Washington, DC, USA, 2007. IEEE Computer Society.

[70] W. D. Yu, P. Supthaweesuk, and D. Aravind. Trustworthy web services based on testing. In *SOSE '05: Proceedings of the IEEE International Workshop*, pages 167–177, Washington, DC, USA, 2005. IEEE Computer Society.

[71] A. Kiežun, Ph. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[72] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[73] G. A. Di Lucca, A. R. Fasolino, M. Mastroianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*, pages 71–80, Washington, DC, USA, 2004. IEEE Computer Society.

[74] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 91–100, New York, NY, USA, 2010. ACM.

[75] E. Bertino and E. Ferrari. Secure and selective dissemination of xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331, 2002.

[76] W. Lee, A. Squicciarini, and E. Bertino. Vulnerabilities leading to denial of services attacks in grid computing systems: a survey. In *CSIIRW '10: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–3, New York, NY, USA, 2010. ACM.

[77] U. K. Tupakula and V. Varadharajan. A practical method to counteract denial of service attacks. In *ACSC '03: Proceedings of the 26th Australasian computer science conference*, pages 275–284, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[78] M. Schillo, H.-J. Bürckert, K. Fischer, and M. Klusch. Towards a definition of robustness for market-style open multi-agent systems. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 75–76, New York, NY, USA, 2001. ACM.

[79] A. Gorbenko, A. Romanovsky, V. Kharchenko, and A. Mikhaylichenko. Experimenting with exception propagation mechanisms in service-oriented architecture. In *WEH '08: Proceedings of the 4th international workshop on Exception handling*, pages 1–7, New York, NY, USA, 2008. ACM.

[80] B. A. Kuperman, C. E. Brodleyd, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detection and prevention of stack buffer overflow attacks. *Commun. ACM*, 48(11):50–56, 2005.

[81] F.-H. Hsu, F. Guo, and T. Chiueh. Scalable network-based buffer overflow attack detection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 163–172, New York, NY, USA, 2006. ACM.

[82] M. Refai. Exploiting a buffer overflow using metasploit framework. In *PST '06: Proceedings of the 2006 International Conference on Privacy, Security and Trust*, pages 1–4, New York, NY, USA, 2006. ACM.

[83] G. Tóth, G. Kőszegi, and Z. Hornák. Case study: automated security testing on the trusted computing platform. In *EUROSEC '08: Proceedings of the 1st European Workshop on System Security*, pages 35–39, New York, NY, USA, 2008. ACM.

[84] Damn vulnerable web app. `http://www.dvwa.co.uk/`. Last accessed on 2011-09-20.