



Norwegian University of  
Science and Technology

# Security Testing of Web Based Applications

Gencer Erdogan

Master of Science in Computer Science

Submission date: July 2009

Supervisor: Torbjørn Skramstad, IDI

Co-supervisor: Per Håkon Meland, SINTEF  
Derek Mathieson, CERN

Norwegian University of Science and Technology  
Department of Computer and Information Science



# Problem Description

Web application users and Web application vulnerabilities are increasing. This will inevitably expose more Web application users to malicious attacks. Security testing is one of the most important software security practices, which is used to mitigate vulnerabilities in software. Security testing of Web applications is becoming complicated, and there is still need for security testing methodologies. This indicates that security testing methodologies for Web applications needs attention. The student will contribute in this respect by doing the following:

1. Execute a thorough research among state-of-the-art security testing methodologies for Web applications.
  
2. Elicit a security testing methodology for Web applications based on certain defined criteria. The overall goal is to elicit a security testing methodology that:
  - (a) Formalizes how to detect vulnerabilities in a Web application, and makes the detection process more efficient regarding time spent and the amount of vulnerabilities that are found.
  - (b) Mitigates false-positives during the security testing process.
  
3. Integrate the elicited security testing methodology from point 2 into the SDLC that is being used by the AIS group at CERN. The integration is to be carried out at a proof of concept level.
  
4. Perform a security test on parts of CERN's largest administrative Web application: Electronic Document Handler (EDH), which has approximately 11,000 users at world basis.
  - (a) A Web Vulnerability Scanner must be evaluated and selected to be used in the testing iterations when there is a need for a testing tool.
  - (b) The security testing will be executed in four iterations; two iterations using the new methodology and two iterations using the old methodology. The testing iterations are executed to collect results based on the old and new security testing methodology.
  - (c) Based on the results from the first and second iteration, an evaluation of the new security testing methodology is to be made.

Assignment given: 20. January 2009  
Supervisor: Torbjørn Skramstad, IDI



# Abstract

Web applications are becoming more and more popular in means of modern information interaction, which leads to a growth of the demand of Web applications. At the same time, Web application vulnerabilities are drastically increasing. This will inevitably expose more Web application users to malicious attacks, causing them to lose valuable information or be harmed in other ways.

One of the most important software security practices that is used to mitigate the increasing number of vulnerabilities is security testing. The most commonly applied security testing methodologies today are extensive and are sometimes too complicated with their many activities and phases. Because of this complexity, developers very often tend to neglect the security testing process. Today, there is only a few security testing methodologies developed especially for Web applications and their agile development environment. It is therefore necessary to give attention to security testing methodologies for Web applications.

A survey of state-of-the-art security testing methodologies for Web applications is performed. Based on some predefined criterions, Agile Security Testing is selected as the most adequate security testing methodology for Web applications, and is further extended to support all the predefined criterions. Furthermore, the extended Agile Security Testing methodology (EAST) is integrated into the Software Development Life Cycle applied by the Administrative Information Services group at the Department of General Infrastructure Services at CERN–The European Organization for Nuclear Research. Finally, by using the EAST methodology and the security testing methodology applied by the AIS group (which is an ad hoc way of performing security tests), an evaluation of the EAST methodology compared to existing ad hoc ways of performing security tests is made. The security testing process is carried out two times using the EAST methodology and two times using the ad hoc approach. In total, 9 vulnerability classes are tested. The factors that are used to measure the efficiency is: (1) the amount of time spent on the security testing process, (2) the amount

of vulnerabilities found during the security testing process and (3) the ability to mitigate false-positives during the security testing process.

The results show that the EAST methodology is approximately 21% more effective in average regarding time spent, approximately 95% more effective regarding the amount of vulnerabilities found, and has the ability to mitigate false-positives, compared to existing ad hoc ways of performing security tests.

These results show that structured security testing of Web applications is possible not being too complicated with many activities and phases. Furthermore, it mitigates three important factors that are used as basis to neglect the security testing process. These factors are: The complexity of the testing process, the “too time-consuming” attitude against security testing of Web applications and that it’s considered to lack a significant payoff.

# Preface

This Master thesis concludes my MSc in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The thesis has been written as a part of a Technical Student program at CERN, the European Organization for Nuclear Research, in Geneva, Switzerland, in the period from January 2009 to July 2009. The work has been carried out in the GS-AIS-EB section at CERN. The thesis consists of:

- A study of state-of-the-art security testing methodologies for Web applications, in which one security testing methodology is elicited based on defined criterions.
- The integration of a security testing methodology for Web applications into a software development life cycle (SDLC), at a proof of concept level.
- An evaluation of the integrated security testing methodology by: (1) performing security tests using the integrated security testing methodology, (2) performing security tests using existing ad hoc ways of performing security tests, and (3) comparing the test results obtained in point (1) and (2).

Geneva, July 28, 2009.



---

Gencer Erdogan





# Acknowledgments

I would like to express my sincere appreciation to my advisors: Per Håkon Meland (SINTEF ICT) and Professor Torbjørn Skramstad (Department of Computer and Information Science, NTNU) for their guidance, encouragement and continuous support throughout the course of this work.

I would like to express my sincere appreciation to my supervisor Derek Mathieson (Section leader at GS-AIS-EB, CERN) for making it possible to realize my thesis in the GS-AIS-EB section. I would also thank Derek Mathieson for his guidance and continuous support throughout the course of this work. Additionally, I would like to thank Benjamin Couturier and Istvan Kallai (from the GS-AIS-EB section) for showing interest in my work and for sharing their opinions with me.

Last, but certainly not least, I would like to thank my family for their everlasting love, support and encouragement throughout my life and studies—“thanks” would never suffice. I hope to make you proud.



# Contents

<b>1</b>	<b>Thesis introduction</b>	<b>1</b>
1.1	Background and motivation . . . . .	4
1.2	Problem statement . . . . .	6
1.3	Research goals . . . . .	6
1.4	Research method and approach . . . . .	7
1.5	Thesis outline . . . . .	10
<b>2</b>	<b>Security testing of Web applications</b>	<b>13</b>
2.1	Software security and security testing . . . . .	13
2.2	Security testing methodologies . . . . .	16
2.3	Security testing methodologies for Web based applications . . . . .	19
2.3.1	Agile Security Testing . . . . .	19
2.3.2	A penetration testing approach . . . . .	24
2.3.3	The Open Web Application Security Project (OWASP) testing framework . . . . .	27
2.4	Security testing tools . . . . .	36
2.5	Acunetix Web Vulnerability Scanner (WVS) . . . . .	40
2.5.1	Why Acunetix WVS? . . . . .	43
2.6	Relevant competitors to Acunetix WVS . . . . .	45
2.7	Comparison, evaluation and selection . . . . .	49
2.7.1	Selection justification . . . . .	50
<b>3</b>	<b>Current situation</b>	<b>55</b>
3.1	CERN - The European Organization for Nuclear Research . . . . .	55
3.1.1	The Administrative Information Services (AIS) group . . . . .	58
3.2	The Scrum methodology: A short introduction . . . . .	59
3.3	The need for a security testing methodology . . . . .	61
3.4	Risk analysis . . . . .	63

---

3.4.1	The CORAS methodology: A short introduction . . . . .	63
3.4.2	Conducting the risk analysis . . . . .	65
<b>4</b>	<b>Contribution</b>	<b>73</b>
4.1	Extending Agile Security Testing . . . . .	73
4.2	Integrating EAST in the AIS group's SDLC . . . . .	77
4.3	Performing security tests . . . . .	79
<b>5</b>	<b>Realization</b>	<b>85</b>
5.1	The integration of EAST: How, Why & Who . . . . .	85
5.2	Security testing of EDH . . . . .	90
5.2.1	The test environment and the testers . . . . .	96
5.2.2	Testing using the current methodology . . . . .	96
5.2.3	Testing using the EAST methodology . . . . .	102
5.3	Test results . . . . .	114
5.3.1	The current methodology test results . . . . .	114
5.3.2	The EAST methodology test results . . . . .	114
<b>6</b>	<b>Evaluation and discussion</b>	<b>117</b>
6.1	Comparison of the methodologies . . . . .	117
6.2	Threats to validity . . . . .	119
6.3	Goals attained . . . . .	121
6.3.1	Problem statement goals . . . . .	121
6.3.2	Research goals . . . . .	122
6.4	The hypothesis: Verified or falsified? . . . . .	124
<b>7</b>	<b>Conclusion and further work</b>	<b>125</b>
7.1	Conclusion . . . . .	125
7.2	Further work . . . . .	126
	<b>List of Figures</b>	<b>129</b>
	<b>List of Tables</b>	<b>131</b>
	<b>Glossary</b>	<b>133</b>
	<b>Bibliography</b>	<b>135</b>
	<b>Appendices</b>	<b>145</b>
<b>A</b>	<b>The current methodology: the guidelines</b>	<b>A-1</b>

# Chapter 1

## Thesis introduction

The increasing number of Web application users indicates that Web applications are becoming more and more popular in means of modern information interaction. In turn, this leads to a growth of the demand of Web applications. This is due to their capability of providing convenient access to information and services in ways not previously possible [68].

In 2007, Rubicon [22] examined the growth of Web applications in the US by surveying over 2000 US adults who had a personal computer at home. This survey revealed that 80% of US home PC users had heard of Web applications, and that 37% of US home PC users used at least one Web application on regular basis (see Figure 1.1). According to Rubicon, the latter result indicates that the usage of Web applications spreads far beyond the 16% of the population traditionally identified as innovators and early adopters. Another interesting discovery was made in Rubicon’s survey: 38% among those who did not use Web applications were worried about security risks and used this as the main reason for why they did not use Web applications (see Figure 1.2). Unfortunately, these people have a good reason to worry about security risks. In their Global Internet Security Threat Report, Symantec reports that they detected 499,811 new malicious code threats during the second half of 2007 [118, 70]. This is a 571% increase over the second half of 2006 (see Figure 1.3). Apparently, the main motivation that drives malicious users to obtain private information by performing attacks such as XSS (cross site scripting) and SQL injection on Web applications (which are some of the most known attack types executed on Web applications [56]), is the economical gain they get by selling what they obtain. The selling is mainly happening through Web based forums and Internet relay chat (IRC) channels, in which they publish advertisements of “their goods” [73]. In their survey of cybercrime activity in the underground economy, Symantec estimates the value

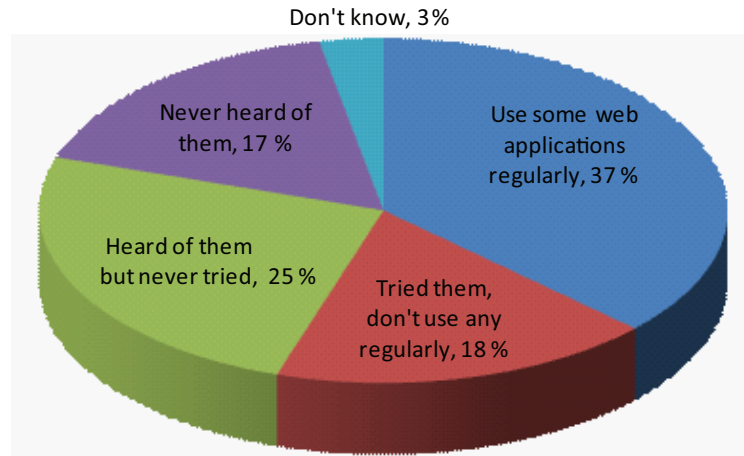


Figure 1.1: *The usage of Web applications among US adults. This figure is adapted from Rubicon [22].*

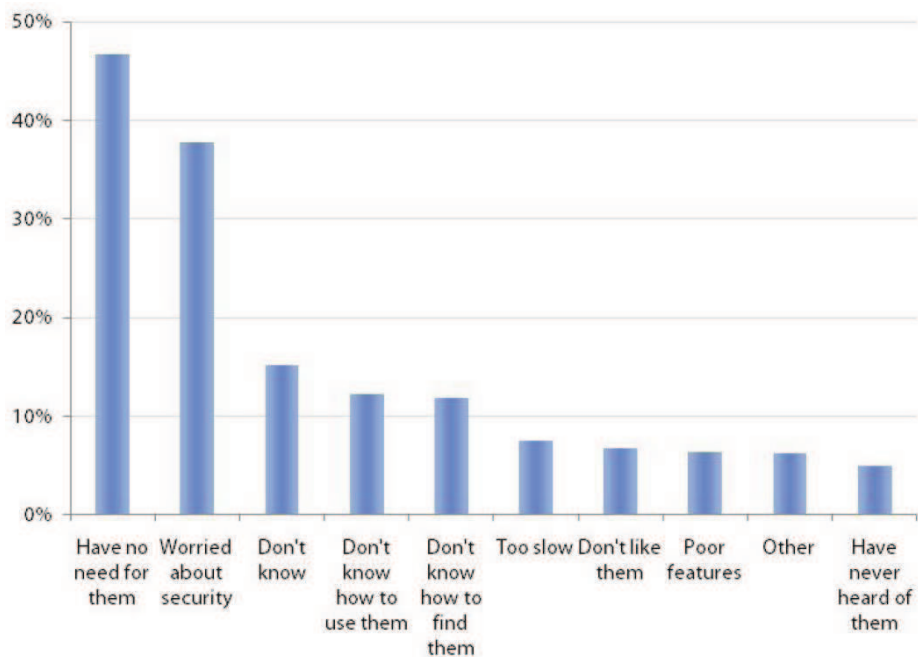


Figure 1.2: *These comments were given by the Rubicon's survey participants as an explanation for why they did not use Web applications. Note that "Worried about security" is the second most common response. This figure is adapted from Rubicon [22].*

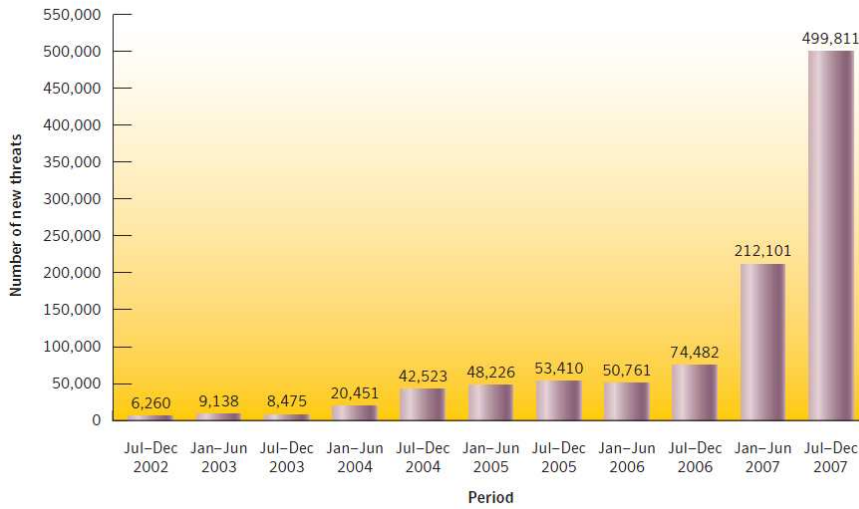


Figure 1.3: *Malicious code threats evolution in the period from July 2002 to December 2007, as given by Symantec [118]. This figure is adapted from Symantec [118].*

of total advertised goods on observed underground economy servers to be over \$276 million for the reporting period (July 2007 - June 2008) [73]. Table 1.1 shows how much of the total value each good covers. A good's percentage value and rank is closely coupled to their level of demand by the "underground market". The fact that Web application vulnerabilities and Web application

Rank	Category	Percentage
1	Credit card information	59%
2	Identity theft information	16%
3	Server accounts	10%
4	Financial accounts	8%
5	Spam and phishing information	6%
6	Financial theft tools	<1%
7	Compromised computers	<1%
8	Malicious applications	<1%
9	Website accounts	<1%
10	Online gaming accounts	<1%

Table 1.1: *Value of advertised goods as a percentage of the total value (\$276 million). Credit card information seems to be the good with the highest value, which is a side effect of being the good of highest demand. This table is adapted from Symantec Report on the Underground Economy [73].*

users are increasing will inevitably expose more people to attacks, which may further cause them to lose information that is of particular value (credit card

information, account credentials, etc). Although the software security field is quite new (some of the first books and academic classes about software security appeared in 2001 [93]), there have been suggested and published a number of software security practices throughout the years. A software security practice is the result of a systematic study for creating secure software [93]. Some of the major software security practices that have evolved through the years are; the Security Development Lifecycle (SDL) by Microsoft [82, 89], the Secure Software Development Lifecycle (SSDL) [121], the Risk Management Framework (RMF) [95], the CORAS methodology [111, 79], and different security modeling techniques as described by Erdogan and Baadshaug [70]. The purpose of these (and other) software security practices is to ultimately mitigate software vulnerabilities. E.g. Web application vulnerabilities.

However, despite the numbers of security practices that have evolved throughout the years, security is often bolted on late in the development and executed as a penetrate-and-patch activity in an ad hoc fashion [57]. In turn, this has a ripple effect on software security practices as a whole. One of the most important software security practices that is affected negatively, as an outcome of this, is software security testing (hereby security testing).

Security testing is an activity that reveals whether the security functionalities are properly implemented, and whether software behaves correctly in the presence of a malicious attack. This is achieved by using the results from an architectural risk analysis, and the results from a risk based security test plan as a *base* for the security testing [96]. In order to benefit from security testing, formal methodologies for performing security tests in terms of who should test and how it should be tested must exist. Further, the formal security testing methodologies must be implemented in the SDLC to make sure that security testing is executed [121].

Security testing gives developers an understanding and overview of: (1) whether the security requirements are fulfilled or not, and (2) which of the vulnerability classes are present in the software (the test object). From the results of these points, developers can deduce facts regarding the security quality of the software. In addition, it increases the security specific knowledge of the developers in means of how software vulnerabilities may be exploited (i.e. thinking like a malicious user), which further makes them more aware of software security next time they develop software.

## 1.1 Background and motivation

The field of security testing has yet to be matured; there is still need for security testing methodologies, techniques, and tools [115]. A security testing method-



ology covers the question of who should test, and how it should be tested, much like in any other form of testing. A security testing technique is used to discover security vulnerabilities at the implementation level (bugs) or at the design level (flaws). A security testing tool is used to discover security vulnerabilities that may have been introduced into the software at the implementation-level (bugs). Security testing has generally been an afterthought in traditional testing [121]. One of the reasons to this is the field's immaturity.

Adding security practises (such as security testing) late in the development increases the probability for vulnerabilities to lie dormant in software for a long time before discovery. In the domain of Web applications, this increases the risk of being exposed to an attack. In addition, the longer vulnerability lies dormant, the more expensive it can be to fix the problem [58, 94] (see Table 1.2).

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

Table 1.2: *The earlier a defect (vulnerability) is uncovered, the cheaper it is to fix. This table is adapted from The Art of Software Security Testing [121].*

Security testing of Web applications has even more factors that make the process complicated: Firstly, Web applications have a very short time-to-market, which is why developers very often tend to neglect the testing process. Secondly, it's considered too time-consuming. And thirdly, it's considered to lack a significant payoff [77, 67, 90].

The result from security tests gives an assurance, to a certain extent, whether the software has an acceptable security quality. It gives significant insight on whether the software should go on production or not. The following facts are present:

1. Web application users and Web application vulnerabilities are increasing.
2. Security testing of Web applications is becoming complicated.
3. There is still need for security testing methodologies.

The above mentioned facts indicate that security testing methodologies for Web applications needs attention.

## 1.2 Problem statement

Based on what has been explained in the introduction of this chapter and in Section 1.1, the student is to:

1. Execute a thorough research among state-of-the-art security testing methodologies for Web applications.
2. Elicit a security testing methodology for Web applications based on certain defined criteria. The overall goal is to elicit a security testing methodology that:
  - (a) Formalizes how to detect vulnerabilities in a Web application, and makes the detection process more efficient regarding time spent and the amount of vulnerabilities that are found.
  - (b) Mitigates false-positives during the security testing process.
3. Integrate the elicited security testing methodology from point 2 into the SDLC that is being used by the AIS group at CERN. The integration is to be carried out at a proof of concept level.
4. Perform a security test on parts of CERN's largest administrative Web application: Electronic Document Handler (EDH), which has approximately 11,000 users at world basis.
  - (a) A Web Vulnerability Scanner must be evaluated and selected to be used in the testing iterations when there is a need for a testing tool.
  - (b) The security testing will be executed in four iterations; two iterations using the new methodology and two iterations using the old methodology. The testing iterations are executed to collect results based on the old and new security testing methodology.
  - (c) Based on the results from the first and second iteration, an evaluation of the new security testing methodology is to be made.

## 1.3 Research goals

**RG.01** Gain knowledge of state-of-the-art security testing methodologies, with a special focus on Web applications.

**RG.02** Find and evaluate security testing methodologies for Web applications.

**RG.03** Implement one security testing methodology for Web applications into

the SDLC that is used by the AIS group at CERN, and evaluate it.

**RG.04** Get an overview of the different security testing tool categories, along with some tool examples for each category (both freeware and commercial).

## 1.4 Research method and approach

According to March et al. [92, 75], there are two main concepts in the realm of Information Technology (IT) research on methodologies, tools, programming languages, development processes, etc. These are:

1. **Behavioral-science:** Behavioral-science has its origin from research methods within natural science research methods. Furthermore, natural science is often known to have two activities; discovery (in which a scientific claim is proposed or generated) and justification (in which a scientific claim is verified or falsified). I.e., natural science research methods try to understand the reality of, for example, a theory or a law [92].
2. **Design-science:** Design-science has the purpose of creating artifacts that serve human purposes. Design-science is technology oriented and has four types of outputs; constructs, models, *methods* and implementations. Furthermore, it consists of two main activities; build (in which an artifact is constructed for a specific purpose) and evaluate (in which an artifact is evaluated for how well it performs) [92].

Although behavioral-science and design-science are quite different, they are also dependent on each other. Design-science produces artifacts. In turn, these artifacts produce results that can be treated by behavioral-science [92].

Design-science research is quite similar to what is defined as the Technology Research Methodology by Stølen et al. [112]. The Technology Research Methodology consists of three main activities and is an iterative process: (1) problem analysis (in which the need for an artifact for the underlying problem is identified), (2) innovation (in which new artifacts are created, or existing artifacts are extended, in order to satisfy the need), (3) evaluation (in which the artifact is proven to support the need). In this context, artifacts are objects manufactured by human beings, e.g. a new algorithm for a computer program, a new SDLC, a new security testing methodology, etc. Furthermore, Stølen et al. [112] define Classical Research as “*research that is focusing on the world around us, seeking new knowledge about nature, space, the human body, the society, etc.*”. This definition is quite similar to what March et al. [92] define as Behavioral-science. Table 1.3 shows the main elements of Classical Research and Technology Research as described by Stølen et al. [112].

	Classical Research	Technology Research
<b>Problem</b>	Need for new theory	Need for new artifact
<b>Solution</b>	New explanations (new theory)	New artifact
<b>Solution should be compared to...</b>	Relevant part of the real world	Relevant need
<b>Overall hypothesis</b>	The new explanations agree with reality	The new artifact satisfies the need

Table 1.3: *The main elements of Classical Research and Technology Research. This table is adapted from Stølen et al. [112].*

The overall goal for this thesis (defined in point 2 in Section 1.2) clearly shows that the purpose of the thesis is to elicit a security testing methodology (the artifact) that makes the security testing of Web applications more reliable and efficient. The purpose is not to understand the reality of e.g. a theory, or to seek new knowledge about nature, space and the human body, etc. (Behavioral-science and Classical Research). Furthermore, Design-science and Technology Research, which are quite similar (see Figure 1.4), are technology oriented. The author has therefore selected Technology Research as the research method and approach for this thesis:

- In the Problem Description phase, the need for a security testing methodology for Web applications is given (the artifact).
- In the Survey of SotA phase, security testing methodologies for Web applications are analyzed and evaluated, and then elicited for a specific purpose. The literature regarding security testing methodologies is mainly obtained from academic books, scientific databases, and domain specific Web sites.
- In the Innovation phase, the elicited security testing methodology is adjusted and implemented in a specific Software Development Life Cycle (at a proof of concept level). The final result is the artifact.
- In the evaluation activity, the artifact is evaluated for its efficiency.

Furthermore, the overall hypothesis of the technology research is: *The artifact satisfies the need* [112]. The overall hypothesis in this thesis is:

**H.00** The detection of vulnerabilities in Web applications is done significantly more efficient regarding time spent, the amount of vulnerabilities that are found and managing false-positives by using a structured security testing methodology for Web applications, compared to existing ad hoc ways of performing security tests.

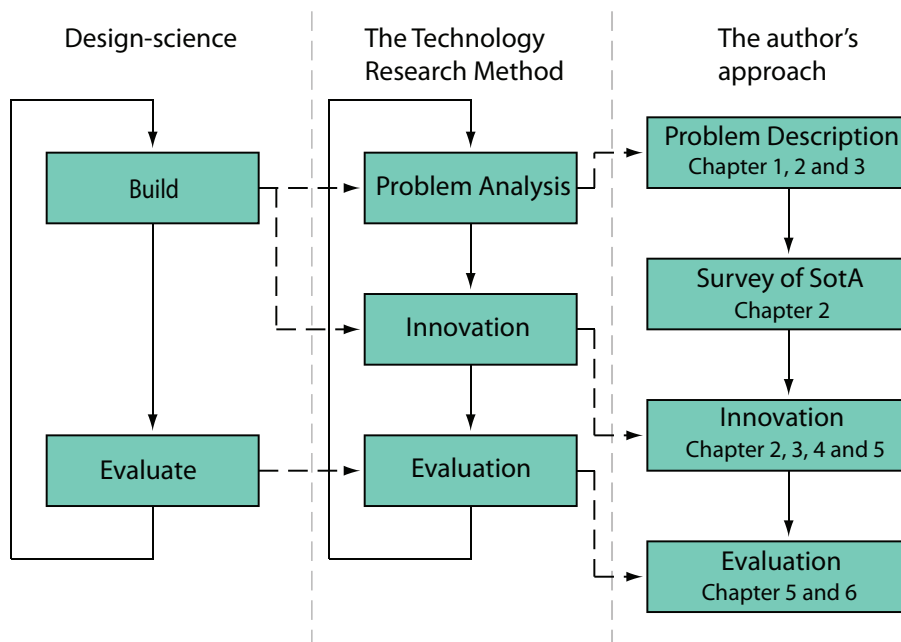


Figure 1.4: *The Design-science approach to research, the Technology Research approach, and the author's approach. The dashed arrows from the Build phase in Design-science to the Problem Analysis and Innovation phases in Technology Research indicates that the latter phases are regarded as a whole in the Build phase. Technology Research explicitly define problem analysis and innovation as two separate phases. In addition to the three phases in Technology Research, the author performs an extra phase, which is a survey of state-of-the-art. Hence, "The author's approach". Both Design Science and Technology Research can be applied iteratively. The figure also shows in which chapter the phases of the author's approach are covered.*

## 1.5 Thesis outline

This Thesis is organized in the following chapters:

### 2 - Security testing of Web applications

This chapter is the result of a thorough research that consists of security testing methodologies with the main focus on state-of-the-art security testing methodologies for Web applications. It explains the connection between software security and security testing, and touches upon general security testing methodologies. The following security testing methodologies for Web based applications are described:

- Agile Security Testing
- A Penetration Testing Approach
- The OWASP Testing Framework

Furthermore, a short introduction to security testing tools and their different categories are given, along with some tool examples. Then, a description of Acunetix Web Vulnerability Scanner (WVS) is given along with an explanation of why Acunetix WVS is selected to be used in the security testing iterations in Chapter 5. A list of Acunetix' relevant competitors are given, which are also the tools Acunetix is evaluated against. Finally, there is a comparison and evaluation among the security testing methodologies listed in the three points above, where one methodology is selected to be used for security testing in the AIS group at CERN.

### 3 - Current situation

This chapter starts by giving a brief introduction of CERN, and the AIS group and its role at CERN. Further, it explains and describes how security testing is currently done at the AIS group, and explains the need for a security testing methodology for Web applications in the group. Finally, some potential threats for the Web applications in the AIS group (EDH is one of the applications) are given along with their related consequences. This is carried out by using the CORAS security risk analysis methodology.

### 4 - Contribution

This chapter describes the author's contributions to security testing of Web based applications, which are:

1. Extending the Agile Security Testing methodology to make it support the criteria defined in Chapter 2.

2. Integrating the extended Agile Security Testing methodology (at a proof of concept level) into the AIS group's SDLC, which is Scrum.
3. Performing security tests on parts of EDH using the extended Agile Security Testing methodology and thereby measuring its efficiency, compared to existing ad hoc ways of performing security tests.

### **5 - Realization**

This chapter describes the realization of the contribution. That is, it describes how the testing methodology is integrated into AIS group's SDLC, and how the security testing iterations are executed, along with their results.

### **6 - Evaluation and discussion**

This chapter presents an evaluation and discussion of:

- The testing process and the results obtained from the tests.
- The threats to the validity of the security test results.
- The project goals and whether they are attained.
- The verification or falsification of the hypothesis defined in Chapter 1.

### **7 - Conclusion and further work**

This chapter concludes the thesis by highlighting the main points and giving a discussion of the achievements. Furthermore, it presents thoughts and suggestions of potential future work and improvements.





# Chapter 2

## Security testing of Web applications

This chapter is the result of a thorough research that consists of security testing methodologies with the main focus on state-of-the-art security testing methodologies for Web applications. It explains how security testing of Web applications is done today. It gives an explanation of the connection between software security and security testing, and touches upon security testing methodologies in general. The security testing methodologies for Web based applications are then evaluated based on given criteria. Furthermore, a short introduction to security testing tools and their different categories are given, along with some tool examples. Then, a description of Acunetix Web Vulnerability Scanner (WVS) is given along with an explanation of why Acunetix WVS is selected to be used in the security testing iterations in Chapter 5. A list of Acunetix' relevant competitors are given, which are also the tools Acunetix is evaluated against.

### 2.1 Software security and security testing

Software security is about engineering software so that it continues to function correctly under malicious attack. The software security field is quite new. Some of the first books and academic classes about software security appeared in 2001, in which they represent a systematic study for creating secure software [93]. As mentioned in Chapter 1, some major software security practices have evolved throughout the years since then. These software security practices are applied and used in software development by being integrated in the various phases of the SDLC. Furthermore, as more and more security practices evolved, it

became apparent that every phase of a SDLC could have its corresponding security phase. By putting together the different software security practices used in the different SDLC phases, new SDLCs specialized on security began to emerge. Some of the biggest contributors in this respect are; the Security Development Lifecycle (SDL) by Microsoft [82, 89], the security touchpoints for a SDLC by McGraw (Cigital Inc.) [95], and the Secure Software Development Lifecycle (SSDL) by Wysopal et al. [121].

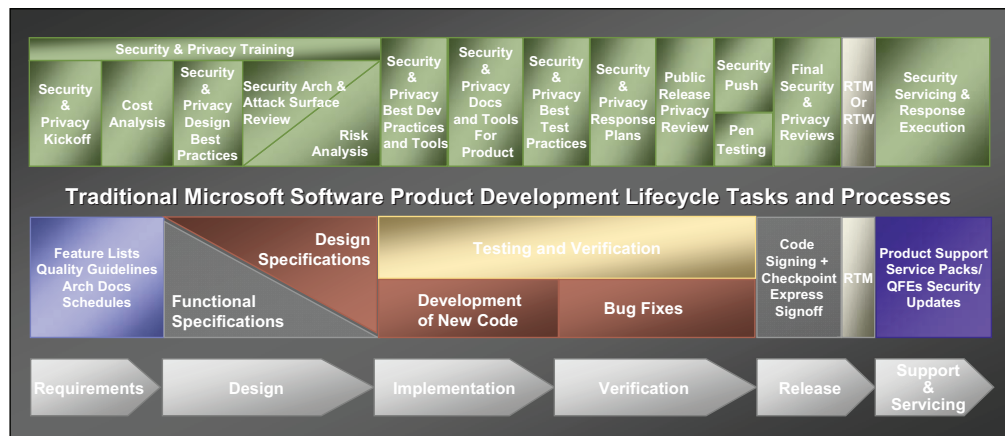


Figure 2.1: The Security Development Lifecycle (SDL) by Microsoft. This figure is adapted from Microsoft SDL V3.2 [97].

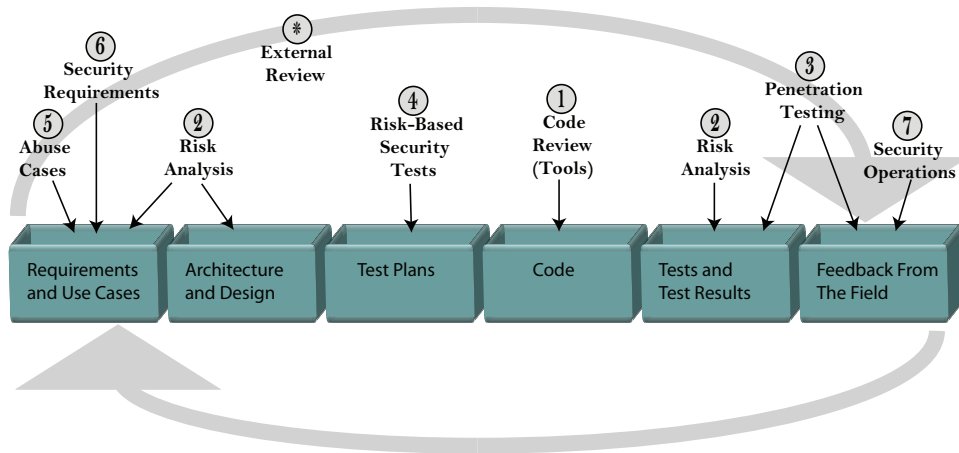


Figure 2.2: The security touchpoints by McGraw (Cigital Inc.). This figure is inspired from McGraw [95].

Figure 2.1 shows the Microsoft Security Development Lifecycle phases (on top of the traditional SDLC used at Microsoft). Figure 2.2 shows the security

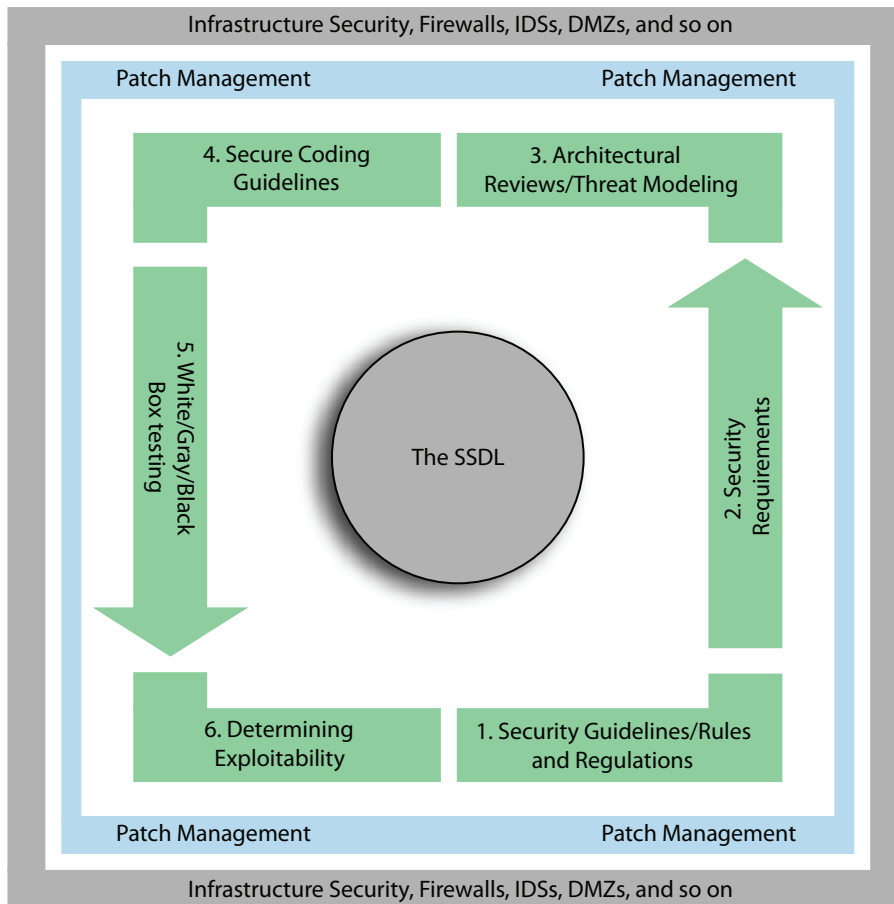


Figure 2.3: *Secure Software Development Lifecycle (SSDL)* by Wysopal et al. This figure is inspired from Wysopal et al. [121].

touchpoints (best practices) by McGraw. The touchpoints are ranked according to their importance, i.e. number 1 being the most important touchpoint. Figure 2.3 shows the Secure Software Development Lifecycle. By comparing these three development life cycles it is possible to see the similarities in their six phases. One obvious fact is that they all are considering software security before, during and after development. Furthermore, the comparing reveals that security testing constitutes a large part in these development life cycles. The security testing practices (reviewing is also a testing practice) are also considered before, during and after development:

- **The Security Development Lifecycle (SDL):**
  - *Before development:* Security requirements reviewing, and security architecture & attack surface reviewing.

- *During development*: Security & privacy best test practices, public release privacy reviewing, pen testing, and final security & privacy reviewing.
  - *After development*: Security servicing & response execution (new patches/updates/builds etc. goes through the same security testing process).
- **The Security Touchpoints:**
    - *Before development*: Security requirements reviewing (touchpoint 5 and 6).
    - *During development*: Risk based security testing (touchpoint 4), code review (touchpoint 1), and penetration testing (touchpoint 3).
    - *After development*: Penetration testing (touchpoint 3) and thereby getting feedback from the field.
- **The Secure Software Development Lifecycle (SSDL):**
    - *Before development*: Security guidelines/rules and regulations reviewing, and security requirements reviewing.
    - *During development*: Architecture reviewing, secure coding guidelines (code reviewing), and white/gray/black box testing (e.g. penetration testing).
    - *After development*: Patch management (managing vulnerabilities); tracking and prioritizing internally and externally identified vulnerabilities, out-of-cycle source code auditing, and penetration testing [121].

In order to say something about the security level in software and to be able to manage the security, security testing must be conducted and carried out before, during and after the development. Security testing reveals whether the security functionalities are properly implemented, and whether software behaves correctly in the presence of a malicious attack. This is why security testing methodologies are needed and why they play such a crucial role in order to achieve secure software.

## 2.2 Security testing methodologies

As shown in Section 2.1, security testing constitutes a large part of the SDLCs specialized on security. Furthermore, these SDLCs show which security testing practices should be carried out in the appropriate SDLC phases (*before, during*

and *after*). By following the security testing practices as given by these SDLCs, structured security testing can be obtained. Each of the SDLCs mentioned in Section 2.1 has thereby a security testing methodology that can be followed.

Another security testing methodology is given by ISECOM [84]; the Open Source Security Testing Methodology Manual (OSSTMM). The OSSTMM is a peer-reviewed methodology for performing security tests and metrics [26]. The guidelines in this extensive testing methodology provides the basis for audits and tools towards:

- A formal scientific method to operational security auditing.
- The metrics to quantify security within any channel.
- The rules of engagement for auditors to assure unbiased and logical analysis.
- A standard for providing certified security audit reports.

It does not only cover security testing of applications, but six “sections” that is related to the application. The six sections are shown in Figure 2.4. The following points list the sections in the OSSTMM, but only one subsection (Internet Application Testing) is expanded to illustrate where in the methodology security testing of Web applications is located. The reader is referred to the OSSTMM [84] for a complete and detailed description of the sections.

1. **Information Security Testing**

2. **Process Security Testing**

3. **Internet Technology Security Testing**

- (a) Internet Application Testing

- i. Re-Engineering
    - ii. Authentication
    - iii. Session Management
    - iv. Input Manipulation
      - Inject SQL language in the input strings of database-tired web applications.
      - Examine “Cross-Site Scripting” in the web applications of the system.
      - etc.
    - v. Output Manipulation
    - vi. Information Leakage

4. **Communications Security Testing**
5. **Wireless Security Testing**
6. **Physical Security Testing**

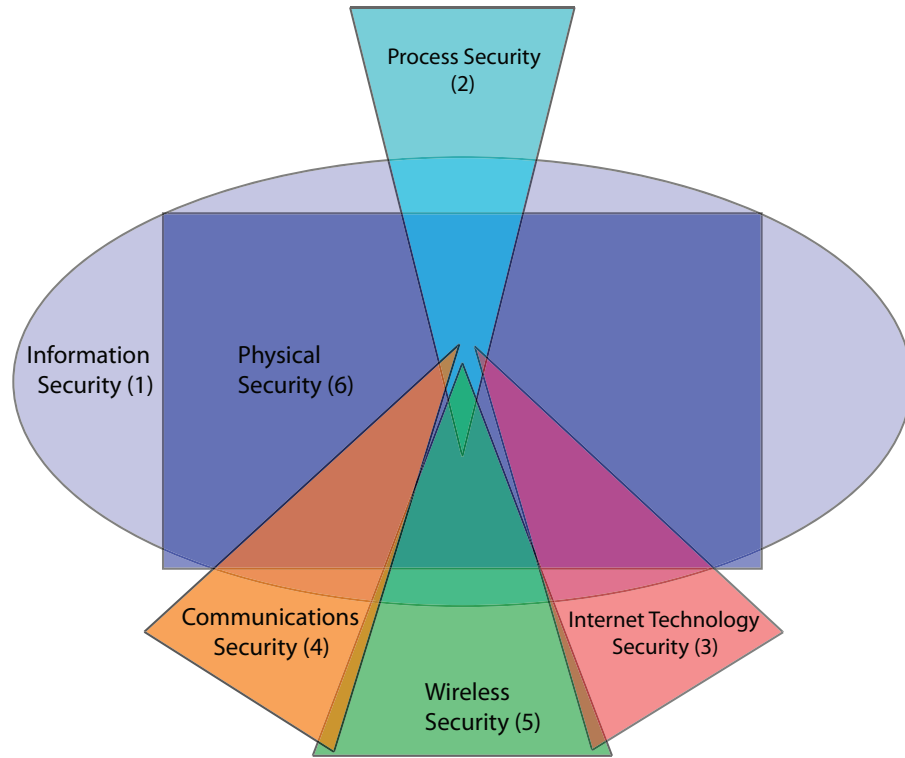


Figure 2.4: The figure illustrates the security testing sections in the Open Source Security Testing Methodology Manual. It also shows that each section overlaps and contains elements of all other sections. This figure is inspired from the OSSTMM [84].

The above mentioned security testing methodologies are based on years of experience and are therefore sound and reliable. However, these testing methodologies are not optimal for every development projects (e.g. Web application development), and can be extensive and very time-consuming. Based on the reasons given in Chapter 1 (i.e. Web application users are increasing and thereby making Web applications the current trend in means of modern information interaction), and because of the scope of this thesis, the focus is set towards security testing methodologies for Web based applications.

## 2.3 Security testing methodologies for Web based applications

The following sections describe security testing methodologies that are especially applicable in Web application development projects.

### 2.3.1 Agile Security Testing

The nature of Web applications requires an iterative and evolutionary approach to development - not a formal rigorous methodology [114]. This has made agile development methodologies the trend for Web application development [85], which has further lead to the idea of agile security engineering. Agile security engineering is the idea of adapting the same philosophy that drive agile software engineering to the traditional practice of mitigating security risks in software [114]. It is a highly iterative process for delivering the security solution and translating security objectives (requirements) into automated security test cases. In addition, it promotes the idea of creating security test cases before the system exists, i.e. test driven development (TDD) (see Figure 2.5), which is another characteristic of agile and iterative development processes [88]. Due to the scope of this thesis, only the testing part of agile security engineering will be regarded, i.e. Agile Security Testing.

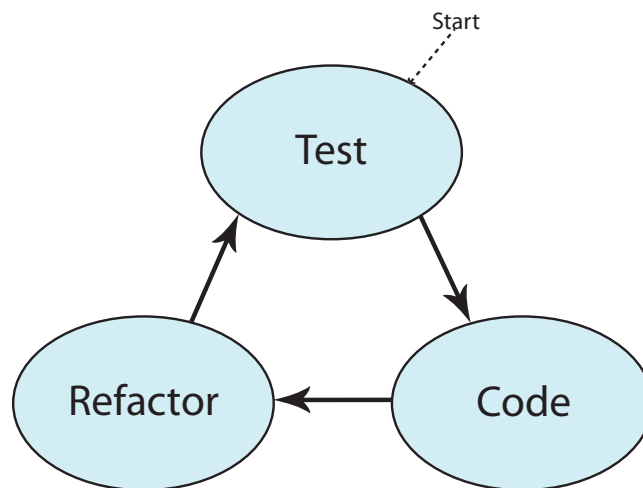


Figure 2.5: *The three steps in TDD. This figure is inspired from TMap Next [88].*

The Agile Security Testing methodology that is suggested by Tappenden et al. [114] consists of the following three main steps:

1. Modeling of security requirements.

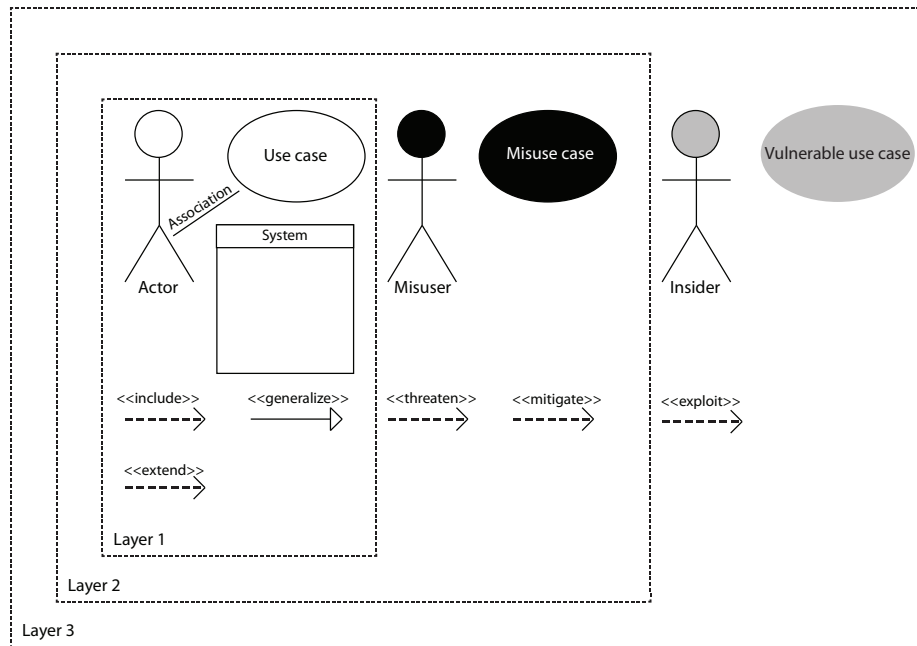


Figure 2.6: *The misuse case legend. Layer 1 shows the UML use case notation, layer 2 shows the misuse case notation [110] and layer 3 shows the extended misuse case notation [105]. Layer 1 is included in layer 2. Layer 1 and layer 2 is included in layer 3. This figure is adapted from Erdogan and Baadshaug [70].*

2. Employing a highly testable architecture.
3. Running automated security tests.

The modeling of security requirements (**step 1**) is executed by creating abuser stories [103, 95] and/or misuse cases [110, 105] in order to elicit security requirements. These are then used as reference points when testing for security in order to verify or falsify a given security requirement. Figure 2.6 shows the misuse case legend. An example of a misuse case diagram is given in Figure 2.7. From this misuse case diagram, it is possible to elicit security requirements such as:

- A client must be identified and authorized with a valid username and password combination to gain access to the Web application.
- The communication between a client and the Web application must be encrypted at all times.
- The network traffic for the Web application shall be monitored for potential denial of service (DoS) attacks.



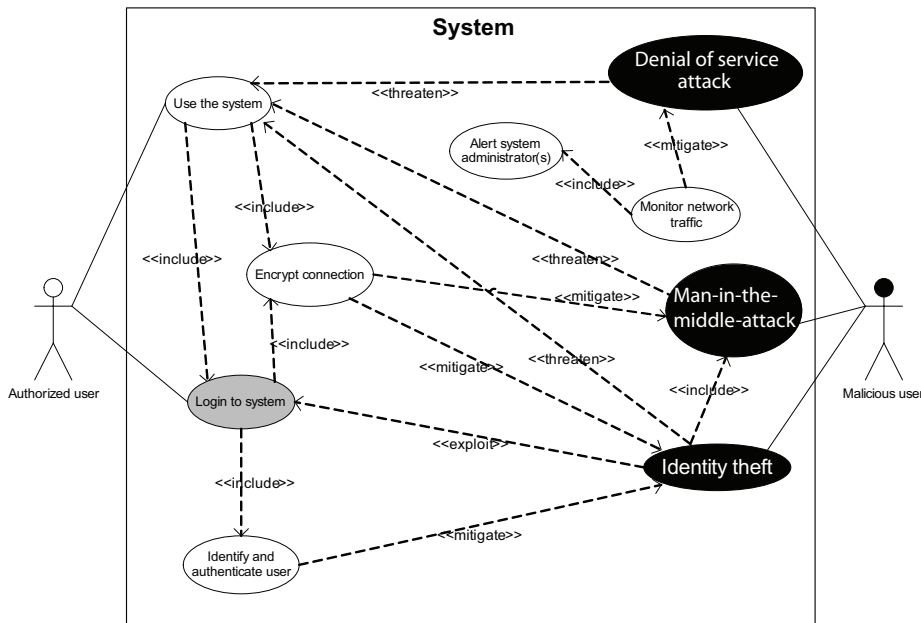


Figure 2.7: An example of a misuse case diagram for a potential Web application. The gray use case denotes a vulnerable use case. The black use cases denote misuse cases. This figure is adapted from Erdogan and Baadshaug [70].

Most Web applications have a three layered architecture; a presentation layer, a business service layer and a data service layer. In more detailed variations of this architecture, the business service layer is further split in two layers - namely a process/control layer and a business entity layer [10]. A highly testable architecture (**step 2**) is achieved by adding a test layer on top of each layer, as shown in Figure 2.8. The resulting architecture suits very well for agile development methodologies because of its many test layers. Additionally, it is useful for security testing because the architecture makes it possible to employ various security testing techniques within any number of the test layers. Furthermore, this architecture makes it possible to carry out the testing in three main levels, which is in line with the well known testing strategy; the V model (see Figure 2.9). Firstly, by creating mock objects, it is possible to execute a single test layer (fits under development tests of the V model). Secondly, by targeting an upper layer and the layer that it depends on, it is possible to execute an integration test, e.g. the process/control layer and the presentation layer (fits under system tests of the V model). Thirdly, by having security requirements as reference points, it is possible to verify or falsify a required security property of the system (fits under acceptance tests of the V model).

In order to fully benefit from Agile Security Testing, the security tests must be automated as much as possible (**step 3**). This can be achieved by using a

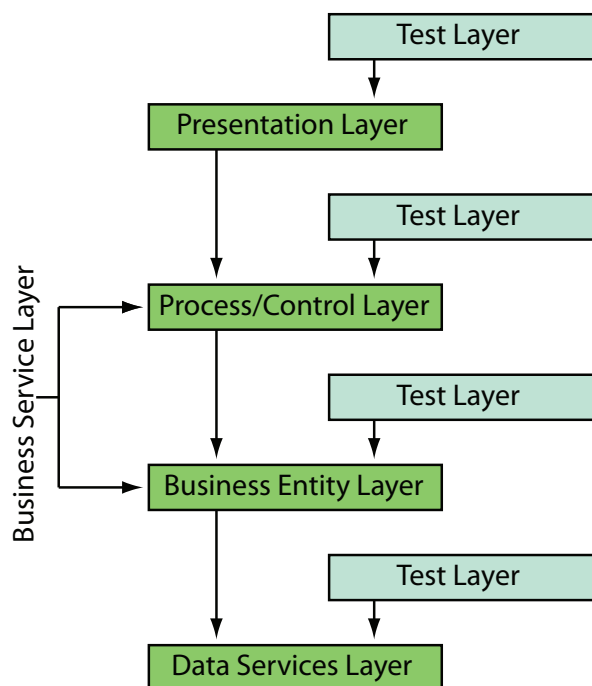


Figure 2.8: A highly testable architecture. This figure is inspired from Tappenden et al. [114].

penetration testing tool, such as Acunetix WVS (explained in Section 2.5).

Agile Security Testing has been used in the industry. An example is Bekk Consulting AS [87], where they approach Agile Security Testing with a methodology possessing the same characteristics mentioned in the three points above. The outcome shows that Agile Security Testing has its benefits and shortcomings. The benefits are:

- *Increased security awareness:* There is a knowledge gap between security experts and software developers [70], and sometimes developers and architects tend to worry less about security [119]. Agile Security Testing brings security testing closer to developers, and thereby increases their security awareness.
- *Collective ownership of security issues:* By integrating security in the development process through misuse cases or abuser stories, and by performing security tests at the development test level, system test level and acceptance test level, security becomes a collective ownership.
- *Easier understanding of security when sufficiently broken down:* By breaking security issues down to misuse cases or abuser stories, developers and architects finds it easier to relate to and to handle.

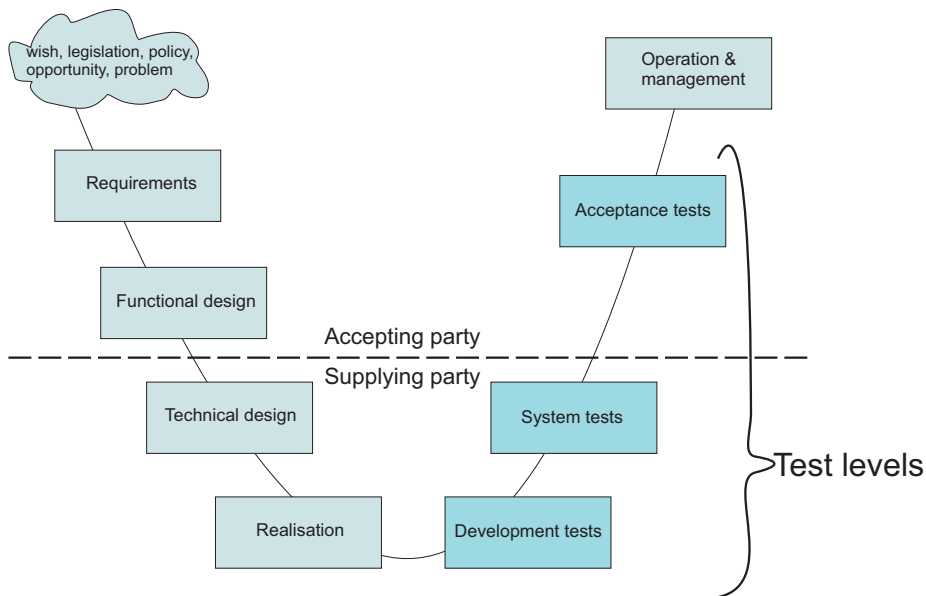


Figure 2.9: *The V model. This figure is inspired from TMap Next [88].*

- *Automatic testing:*
  - Expresses the security requirements with a high degree of formality throughout the acceptance tests.
  - Gives a certain degree of confidence that security regression is not introduced during the project.
- *Getting away from “penetrate and patch”:* This approach to security testing of Web applications promotes the idea of building-security-in the software throughout the development process.

The shortcomings are:

- *Misuse cases/abuser stories incompleteness:* In Agile Security Testing, the misuse cases/abuser stories are developed and handled incrementally together with the system functionality and their related user stories. This, however, does not cover all the security issues because not all misuse cases/abuser stories have their corresponding user story. Consequently, there must be created misuse cases/abuser stories that do not have any corresponding user stories. In order to create such misuse cases/abuser stories, special security knowledge is needed.
- *Test incompleteness:* Not all misuse cases/abuser stories can be expressed as automatic tests. E.g. vulnerabilities that has to be mitigated by hard-

ening the application environment. In these cases, manual tests must be carried out.

- *Security specialist role not eliminated:* The two previous points indicate that there are situations in the security testing process where there is a need for security experts.

### 2.3.2 A penetration testing approach

Penetration testing is the most commonly applied security testing methodology, but it is also the most commonly misapplied security testing methodology [58]. It is misapplied by firstly, being carried out at the end of the SDLC and secondly, by being performed in a “time boxed” manner where a small and predefined portion of time and resources is given to the effort. In turn, this approach uncovers problems too late. Even if the problems are uncovered, fixing them at this stage is prohibitively expensive [58].

In order to prevent the misapplication of penetration testing, Thompson [116] (and Security Innovation [44]) suggests a structured penetration testing methodology. Although this methodology is more formal than Agile Security Testing (see Section 2.3.1), it is applicable for Web application development projects and consists of the following five main steps [116]:

1. Create a threat model
2. Build a test plan
3. Execute test cases
4. Create the problem report
5. Execute a postmortem evaluation

The **first step** in this penetration testing methodology is to create a threat model in order to get a detailed, written description of the risks that threatens the application, which is of the utmost importance to mitigate. Threat modeling [82, 98] is somewhat similar to the misuse case/abuser story approach (see Section 2.3.1) in means of thinking like an attacker when developing the model. The key difference with threat modeling is the ability to get an overview of the various conditions (vulnerabilities) that have to be present in order to realize a given threat. Figure 2.10 shows a partly created threat model (a threat tree) for a Web application that lets users manage their bank account online. A threat tree, and the threat modeling process, helps security testers to break an exploitable threat goal into testable sub goals that they can assess more easily [116].

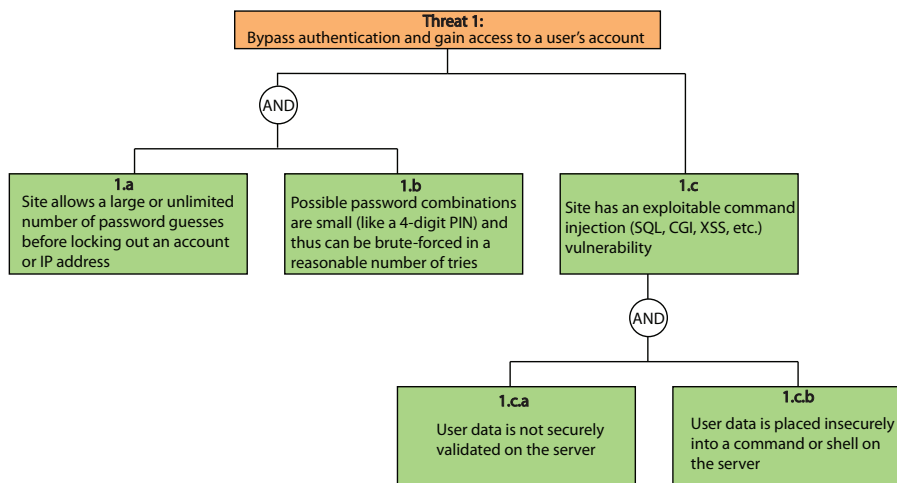


Figure 2.10: An example of a threat model for a Web application that lets users manage their bank account online. In this threat model, only one threat is regarded for such a system; namely “bypass authentication and gain access to a user’s account”. The threat’s child nodes represent the vulnerabilities that have to be present in order to realize the threat itself. The logical AND between the edges of node 1.a and 1.b means that both 1.a AND 1.b have to be present in order to attain the threat. Otherwise the edges are regarded as logical OR. This figure is inspired from Thompson [116].

The **second step** is to build a test plan. The test plan acts as a road map for the total security testing effort. It is created to get a high-level overview of the security test cases, an overview of how exploratory testing (i.e. simultaneous learning, test design, and test execution [60]) will be conducted, and to get an overview of which components will be tested [116]. The test plan must address the following key points:

- *Logistics*: The security testing project schedule and people, resources, and equipment that is needed must be addressed.
- *Deliverables and timeline*: In order to support the development organization to integrate the results into the project timeline, a timetable of activities and a list of deliverables along with their description must be addressed.
- *Test cases and tools*: An overview of the security test cases that will be constructed and executed, the tools that is needed to conduct the tests, and the opportunities for automated testing along with their tools must be addressed.

The **third step** is to execute the test cases. Security issues and insecure

behavior in software is often hard to understand, and thereby makes it challenging to create good security test cases. Fortunately, there exist many extensive public vulnerability databases and mailing lists that are available online. Some of the most known are; the BugTraq mailing list [7], the Computer Emergency Readiness Team (CERT) at Carnegie Mellon University [13], and the Common Vulnerabilities and Exposure (CVE) website [12]. These are invaluable resources for security testers. Furthermore, the security testing of vulnerabilities are divided into four main testing groups [120]:

- *Dependency testing*: Tests in this group has the purpose of uncovering security vulnerabilities in the file system, the registry and external libraries by either denying the application access to these resources, or by tampering with and corrupting data.
- *User interface testing*: Tests in this group has the purpose of uncovering security vulnerabilities that may be exposed through the user interface by generating corrupt and unanticipated input data, e.g. SQL injection and XSS.
- *Design testing*: Tests in this group has the purpose of uncovering security vulnerabilities that may be exposed because of design errors such as unsecured ports and default accounts. Such design errors are often referred to as *flaws* [95].
- *Implementation testing*: Tests in this group has the purpose of uncovering security vulnerabilities that may be exposed because of implementation errors such as time-of-check-to-time-of-use (TOCTOU) and incorrect or none input validation. Such implementation errors are often referred to as *bugs* [95].

The **fourth step** is to create a report of the findings from the security testing process. This is critical for proofing that a given vulnerability is present in the application, at a later hand. The security test report must at least cover the following three points:

- *Reproduction steps*: A complete and unambiguous list of how the vulnerabilities were discovered must be provided. This is necessary in order to pin point and track down how a given vulnerability was exploited. Without this, it is unlikely that the security bug will be fixed. Additionally, a list of reproduction steps makes it possible for other testers and developers to perform the same test and verify the existence of the vulnerability. Finally, information about the environment must also be added. At the very last, the version of the operating system, application, relevant protocols, client

and how they are configured must be provided. This is because different environments may act differently on a given vulnerability attack.

- *Severity*: Severity is based on the potential result of a security failure. An accurate severity ranking is crucial in order to prioritize the most important security features of the application, and making sure that they are present. Failing to do this correctly may cause big problems for the application and its stakeholders, e.g. economic and reputation loss.
- *Exploit scenarios*: An exploit scenario is the specific sequence of things an attacker can do to take advantage of a given vulnerability and the consequences of doing so. This plays a critical role in describing the vulnerability's impact for decision makers.

The **fifth step** is to execute a postmortem evaluation. A postmortem evaluation is basically a meeting session executed by the security test team for analyzing the security bugs/flaws that were detected during the testing process. The focus in a postmortem evaluation should be on why vulnerabilities (bugs or flaws) were missed during development, and how to improve the process to prevent or isolate such security issues in the future. Postmortem evaluations can also help a security testing team to put light on potential holes in the testing process, and sharpen techniques to find certain security vulnerabilities. Furthermore, a postmortem evaluation should be executed after a security testing is completed, while the security issues are still fresh in the testers' minds. It is necessary to mention that postmortem evaluations can be executed iteratively, for example after each security testing iteration.

### 2.3.3 The Open Web Application Security Project (OWASP) testing framework

The Open Web Application Security Project (OWASP) is an open, not-for-profit, community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security [33].

One of the major contributions by OWASP is their security testing framework for Web applications. The framework is not developed for a specific SDLC, but is rather a comprehensive generic development model (see Figure 2.11) that contains the necessary activities needed for systematic security testing of Web applications. Because of the framework's generality, organizations can pick and integrate those activities that fit their development environment and SDLC in

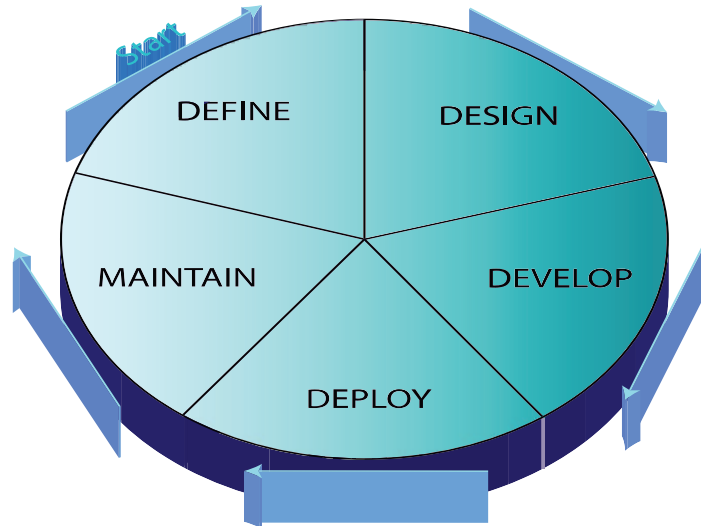


Figure 2.11: *The generic SDLC model. This figure is inspired from the OWASP Testing Guide [102].*

order to develop secure software. The OWASP Testing Framework consists of five main phases [102]:

#### 1. Before development begins

- (a) *Review policies and standards:* This is to ensure that appropriate policies, standards, and documentation are in place and available for the development teams. Further, this gives development teams guidelines and policies that they can follow.
- (b) *Develop measurement and metrics criteria (ensure traceability):* By defining measurement and metrics criteria, and using them throughout the project, provides visibility into defects in both the process and the product.

#### 2. During definition and design

- (a) *Review security requirements:* It is important to have unambiguous requirements. This activity is about testing (reviewing) the assumptions that are made in the security requirements, and reviewing to see if there are gaps in the security requirements definitions. When looking for these gaps, the following security properties should be considered:
  - i. User Management (password reset etc.)
  - ii. Authentication



- iii. Authorization
  - iv. Data Confidentiality
  - v. Integrity
  - vi. Accountability
  - vii. Session Management
  - viii. Transport Security
  - ix. Tiered System Segregation
  - x. Privacy
- (b) *Review design and architecture:* Design and architecture documents (such as models describing the architecture, and their corresponding textual descriptions) must be reviewed to ensure that the design and architecture enforce the appropriate level of security as defined in the requirements. Identifying security flaws in the design phase is one of the most cost-efficient places to identify flaws (see Table 1.2). Additionally, it can be one of the most effective places to make changes. For example, if the architecture reveals that authorization is to be made in multiple places, then it might be more efficient to create a central authorization component. A Comprehensive Method for Architecture Evaluation (ATAM) [61], an architectural review methodology by Maranzano et al. [91] and the Independent Software Architecture Review methodology (ISAR) [113] are some examples of architecture review methodologies. These methodologies can easily be used for security reviewing, as well as other quality features.
- (c) *Create and review UML models:* After creating Unified Modeling Language (UML) models that describe how the application works, they must be reviewed in order to discover security weaknesses. To get an exact understanding of how the application works the UML models must be confirmed with the system designers.
- (d) *Create and review threat models:* After the design and architecture review, along with the UML models explaining exactly how the system works, a threat modeling exercise must be performed. By developing realistic threat scenarios, and analyzing the design and architecture accordingly, makes it possible to look for and ensure that the threats have been mitigated. Additionally, this analysis makes it possible to discover threats that don't have any mitigation strategies. Such discoveries can then be used as reference points to modify the design or architecture in order to mitigate the given threat. Figure 2.10 shows an example of a threat model.

An important contributor to formal security reviewing practices is SHIELDS. SHIELDS is an FP7 (the EU’s Seventh Framework Programme for research and technological development) project concerned with model-based detection and elimination of software vulnerabilities [46]. The main objective of SHIELDS is to bridge the gap between security experts and software developers and thereby reduce the occurrence of security vulnerabilities [71].

SHIELDS suggest goal-driven security inspections and vulnerability-driven security inspections that can be used to review software artifacts, i.e. requirements, design, source code and manuals. The purpose of the goal-driven security inspections is to check that a given security goal has been correctly implemented. The purpose of the vulnerability-driven security inspections is to check whether a vulnerability is present or not in the software artefacts [71]. Furthermore, the goal-driven and vulnerability-driven security inspections have their respective techniques [71]:

- Goal-driven security inspections:
  - *Security Goal Indicator Tree (SGIT)*: Provides a reusable, structured and detailed description of indicators that provide evidence (not necessarily conclusive) of the correct (or incorrect, missing) realisation of a security goal.
  - *Indicator specialisation tree*: Provide details on how to inspect for a given indicator in different software artifacts.
  - *Guided checklist*: Derived from SGITs, and provides an easy to use guide for how to inspect whether a security goal has been correctly implemented.
- vulnerability-driven security inspections:
  - *Vulnerability Inspection Diagram (VID)*: Provides procedural instructions on how to review a software artefact for detecting the presence of a specific class of vulnerability.
  - *Security inspection scenarios*: Ease and increase efficiency of the manual security inspection process through giving detailed description of the security problem and the threat related to it and giving detailed instructions on how to find instances of the vulnerability class in software artefacts.

Figure 2.12 shows an example of an SGIT for the security goal “audit data generation”. Figure 2.13 shows an example of a VID where the vulnerability class is “Revealing Internal Error Message”. Further detailed explanation of the goal-driven and vulnerability-driven security inspections

can be found in SHIELDS deliverables D1.2 [71], SHIELDS deliverables D4.1 [72], and Erdogan and Baadshaug [70].

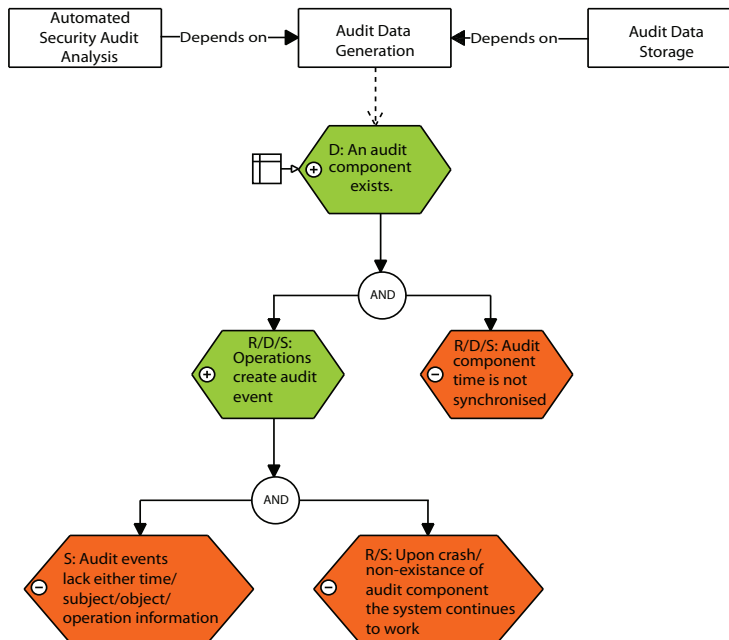


Figure 2.12: An SGIT of the security goal “audit data generation”. The initials in the diagram elements have the following meaning: *R* = Requirements, *D* = Design and *S* = Source code. This figure is inspired from SHIELDS deliverables D4.1 [72].

### 3. During development

- (a) *Code walkthroughs*: The security team should perform a code walkthrough with the developers and system architects in order to get a high-level understanding of the flow, the layout, and the structure of the code that makes up the application. A code walkthrough is a high-level walkthrough of the code where the developers can explain the logic and flow of the implemented code. It allows the code review team to obtain a general understanding of the code, and allows the developers to explain why certain things were developed the way they were [102, 99].
- (b) *Code reviews*: After a code walkthrough, the security team has a better understanding of the code structure and thereby a good starting point of reviewing the code for security defects. An example of a process for performing security code reviews is suggested by Howard [81], in which he explains three main steps for performing a security code

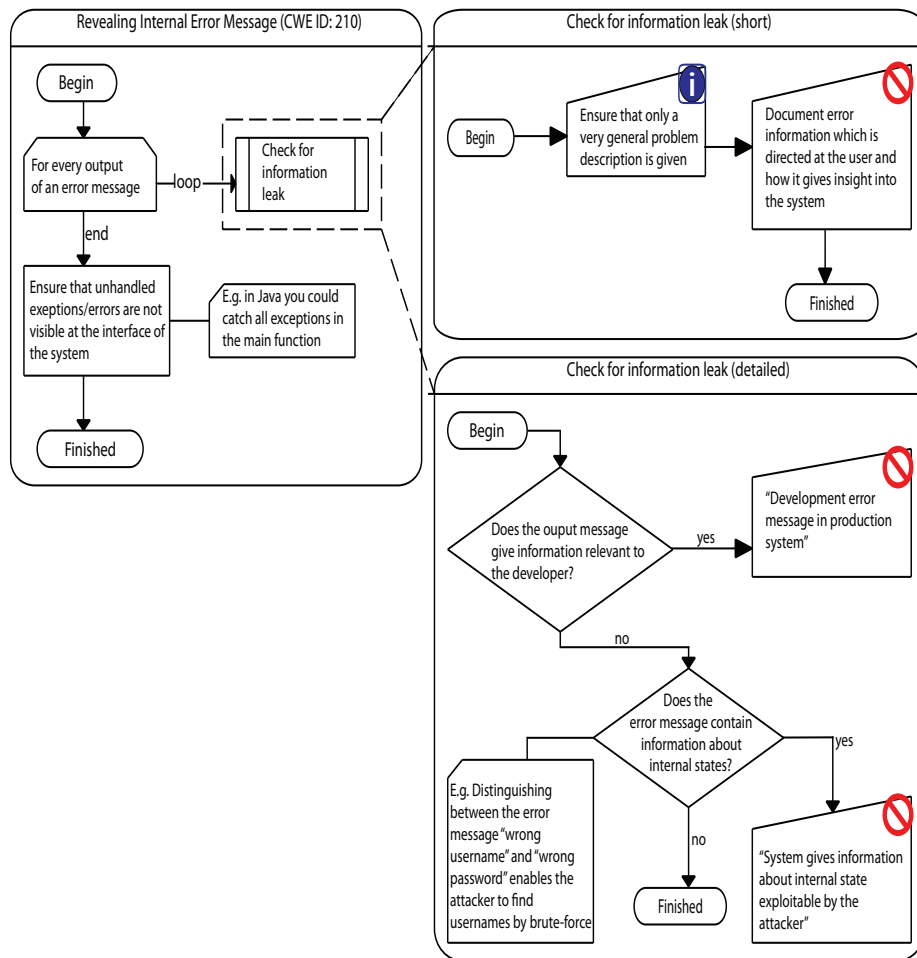


Figure 2.13: A VID example of the vulnerability class; “Revealing Internal Error Message”. The dashed arrows point to the short and detailed version of the “Check for information leak” procedure call. The short version of a procedure call is aimed at security experts and inspectors that have experience in security inspection. The detailed version of a procedure call is a step-by-step guide of how to carry out the VID and is primarily aimed at unexperienced inspectors. This figure is inspired from SHIELDS deliverables D4.1 [72].

review in a structured manner. Briefly explained, the three steps in the process are:

- i. **Make sure you know what you’re doing:** This step is about getting educated on the security issues and to learn about the vulnerabilities and how they appear in software.
- ii. **Prioritize:** This step is about prioritizing which files to review in respect to level of threat, risk, budget, etc. The point is to

review the files (the part of the system) that have the highest risk of being attacked.

- iii. **Review the code:** The final step is to carry out the code review by; (1) rerunning all available code-analysis tools, (2) looking for common vulnerability patterns and (3) digging deeper into risky code.

In addition to these steps, the process provides a graphical modeling technique for modeling charts to guide security testers in the review process. Figure 2.14 shows an example of a chart containing the high-level steps required to identify common XSS issues.

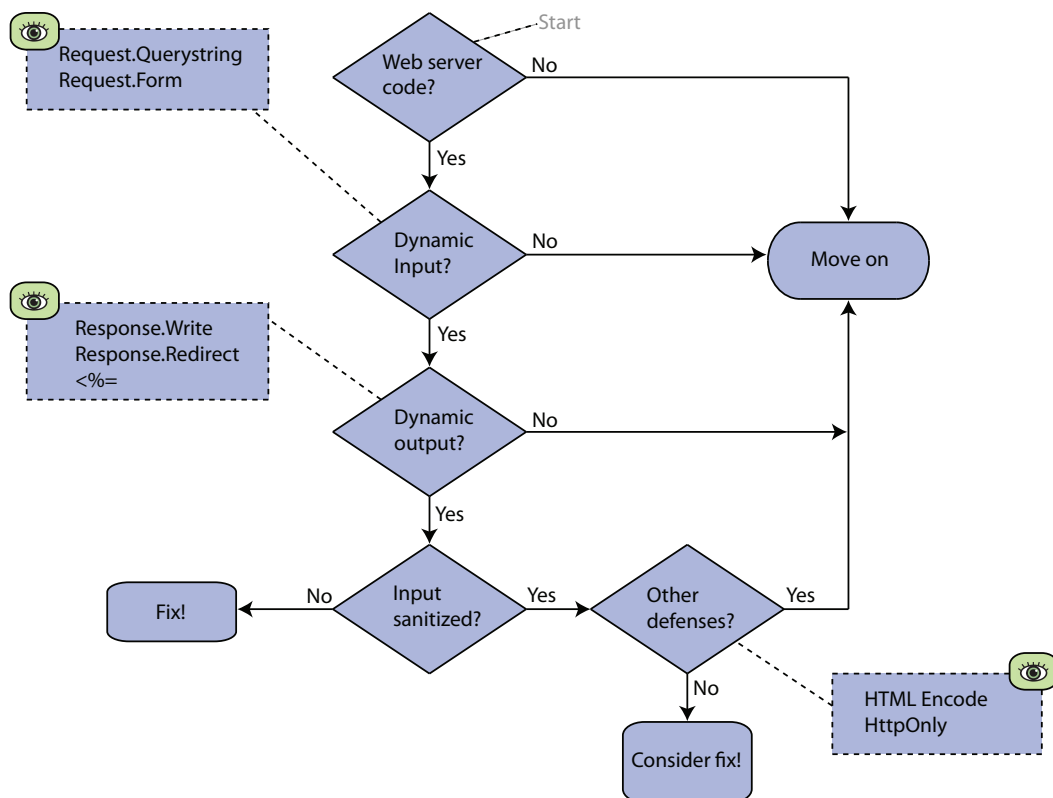


Figure 2.14: Reviewing for cross site scripting (XSS) issues. This figure is inspired from Howard [81].

#### 4. During deployment

- (a) *Application penetration testing:* Having carried out reviews of the security requirements, analyzed the design and architecture for secu-

rity flaws, and performed security code reviews, a penetration testing (explained in Section 2.3.2) should be carried out to make sure that nothing has been missed (i.e. none of the *known* security issues).

- (b) *Configuration management testing*: The application penetration testing process should include the checking of how the infrastructure was deployed and secured. A small aspect of the configuration may be at a default install stage and vulnerable to exploitation.

## 5. Maintenance and operations

- (a) *Conduct operational reviews*: There needs to be a process in place which details how the operational side of both the application and infrastructure is managed [102].
- (b) *Conduct periodic checks*: In order to ensure that no new security risks have been introduced and that the level of security is still sound, health checks of the application *and* the infrastructure should be performed monthly or quarterly.
- (c) *Ensure change verification*: It is vital to check that the level of security is still sound after each new deployment in the production environment, and that the new build hasn't affected the security negatively.

Figure 2.15 shows the OWASP Testing Framework work flow. It gives an overview of the five phases described above, and lists the main activities in each phase.

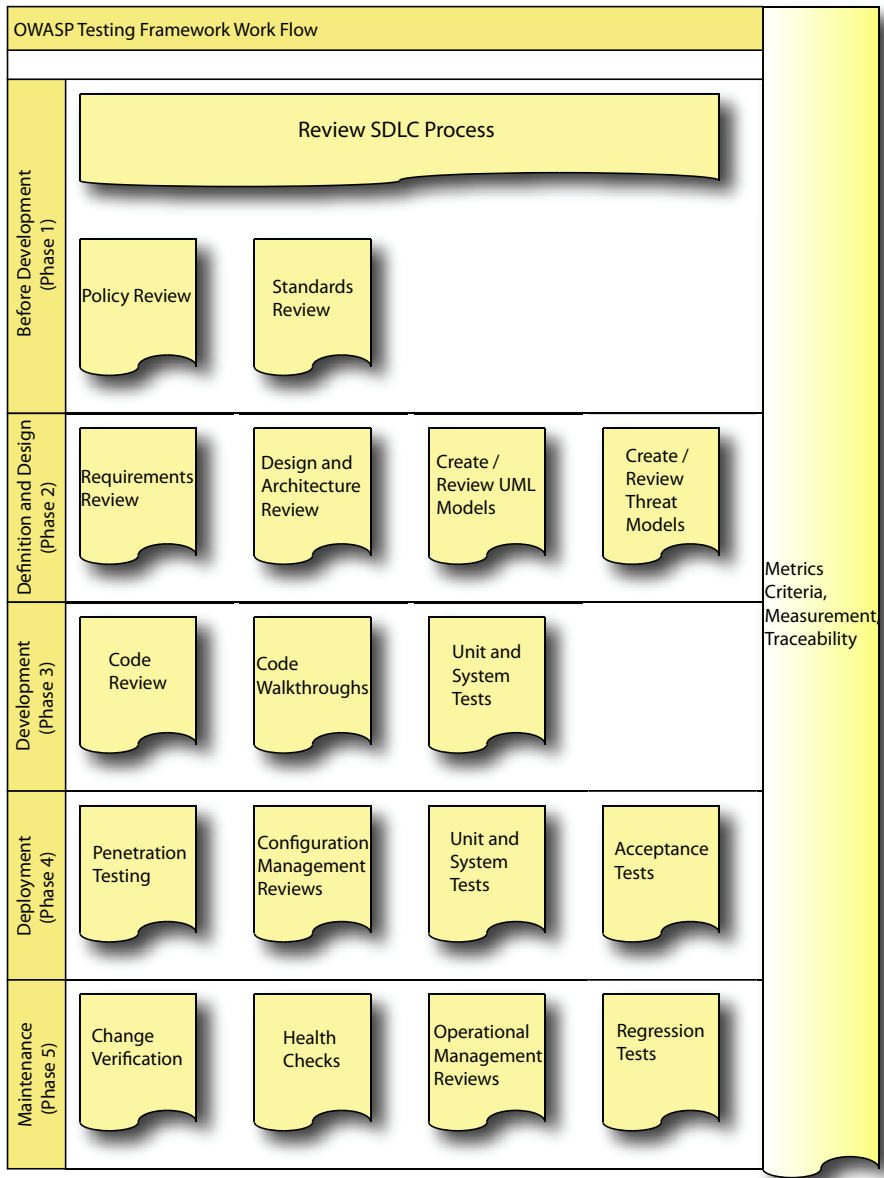


Figure 2.15: The OWASP Testing Framework work flow. This figure is inspired from the OWASP Testing Guide [102].

## 2.4 Security testing tools

Security testing tools provide the automation of security tests. Automated security testing is necessary in order to carry out security tests efficiently, which can sometimes be very extensive [117]. Additionally, for reasons given in Chapter 1 (i.e. significant increase of vulnerabilities in Web applications, Web application's short time-to-market, and that security testing of Web applications is considered too time-consuming) there is an obvious requirement for effective methods to find vulnerabilities in Web applications [66].

Security testing tools can be used to help identify potential security vulnerabilities within commercial and proprietary based software [104]. As Figure 2.1, 2.2 and 2.3 shows, security testing tools are used in both pre-deployment and post-deployment. The focus on the software security field these recent years has also made the software security community to look closer at security testing tools, and the automation of security tests. As a result, there have been produced hundreds of security testing tools (both freeware, and commercial) [45, 34]. These security testing tools can further be divided into several categories with their specific purpose. However, some security testing tools overlap in different categories because they are multi functional. Having so many security testing tools (in different categories) does not make things easier when selecting tools for software development or testing projects. Fortunately, there have been suggested a list of evaluation criteria by Radosevich and Michael [104], and Dustin et al. [69] that can be considered when selecting a security testing tool. The latter evaluation criteria list can be used for any type of testing tools as well as for security testing tools, and is available on the UML Web site [50]. Not all of the criteria in these lists are relevant to all software development or testing projects [104]. It is therefore possible to select some relevant criteria and use them as a starting point for evaluating security testing tools.

The following points explain the different categories of security testing tools, as given by Curphey et al. [63]. Table 2.1 gives some examples of commercial and free tools for each of the categories, and Table 2.2 shows the SDLC phase a tool category is typically associated with.

- **Source-code analyzers** are divided in two sub-categories; static analyzers and dynamic analyzers. *Static analyzers* run text-based searches for strings and patterns in source code files and tries to find security defects in the source code, and then generates a report of the findings. They are generally fast, because they require very little processing. On the other hand, they are quite limited in what they actually find and have a tendency to produce false-positives. *Dynamic analyzers* run a deeper analytical analyze of the source code. They don't only find instances of



bad coding practices, but also perform rich control and dataflow traversals. This reduces the false-positive rate dramatically, but at the same time makes them slow and performance intensive.

- **Web application (black-box) scanners** perform security tests on Web applications by (usually) first crawling through the entire Web site that's holding the Web application, and then running specific security test cases wherever possible. All the tests are performed over the HTTP protocol. They are not only effective at finding attack incidents like SQL injection and cross-site scripting, but also at finding configuration management issues (especially those related to Web servers). These tools are usually not aimed at developers, this makes the mitigation process complex.
- **Database scanners** are used to perform various database queries to analyze the database's security configuration. It also scans the database configuration to determine common problems (e.g. loosely permissioned and powerful stored procedures). Additionally, they verify database users and role membership against known best practices.
- **Binary analysis tools** usually look for security issues in two steps. They first determine the public interface of the application or library. Then, they attempt to "fuzz" the input parameters and look for signs of an application crash, common vulnerability symptoms, and other unusual behavior. Fuzz testing is a type of blackbox random testing which randomly change well-formed inputs and tests the application on the resulting data [74]. Some tools under this category (e.g. FxCop, see Table 2.1) use reflection and introspection to find security issues in application binaries. Although these applications are easy to use, they tend to produce very complex results that require highly skilled analysts to understand the results. Additionally, these tools have an extremely high false-positive rate [63].
- **Runtime analysis tools** are much like dynamic source-code analyzers, but the key difference is that they don't identify application bugs, but instead give reviewers and testers a variety of critical information, e.g. application control flows and data flows.
- **Configuration analysis tools** are used for static analysis of the application configuration files, host settings, or Web/application server configuration. They are most useful for deployment-security evaluation and for ensuring that the application operates under desired security context. They can also analyze files such as `web.xml`, `machine.config` and `web.config` for security issues.

- **Proxy tools** are used between a client and a Web server that hosts a Web application. They allow testers to trap the HTTP request that leaves a client or the HTTP response that leaves a Web server. Furthermore, they let testers view and modify different parts of the request, e.g. cookies, HTTP headers, GET and POST parameters and HTML content. They can also be used to efficiently bypass client side validation.
- **Miscellaneous tools:** Some tools don't fit under a specific category because they are multi functional, e.g. Visual Studio Team System (VSTS) that is a mix of source code analyzer, load/stress testing tool, web application (black-box) scanner and unit testing tool.

In addition to the above mentioned categories there exist other security testing tool categories that are defined by OWASP (currently without description) [35]:

- Application Vulnerability Scanning Tools (*Subcategory of Web application (black-box) scanners?*)
- Application Penetration Testing Tools (*Subcategory of Web application (black-box) scanners?*)
- Test and Educational Applications (*Subcategory of mixed tool types (miscellaneous tools)?*)
- Application Security Analysis Support Tools (*A collective term for source-code, binary, runtime and configuration analysis tools?*)

Furthermore, insecure.org [45] distinguish between vulnerability scanners and Web scanners, and have named other categories such as password crackers, sniffers, and packet crafters. In addition to automated security testing, there exist tools that actually create security test cases based on models (e.g. UML models and UMLsec models) [107, 86].

All these different, and sometimes overlapping, categories of security testing tools indicate the difficultness of tool categorization and the lack of standards for software security testing tools.

Tool type (category)	Commercial	Free/Open Source
Source-code analyzers	Fortify SCA [21], Klockwork Insight [28]	Rough Auditing Tool for Security [41], FindBugs [19]
Web application (black-box) scanners	Acunetix WVS [2], HP WebInspect software [23]	WebScarab [37], Burp Suite [8]
Database scanners	AppDetective [4], NeXpose [32]	Scuba [42], SQLRecon [48]
Binary analysis tools	IDA Pro [25], BinNavi [5]	BugScam [6], FxCop [29]
Runtime analysis tools	Rational Purify-Plus [40], Compuware BoundsChecker [14]	.NETMon [31], CLR Profiler [11]
Configuration analysis tools	Desaware CAS/Tester [17]	SSLDigger [49], Security Configuration Tool set [30]
Proxy tools		Paros [38], Fiddler [18]
Miscellaneous tools	Visual Studio Team System [53]	Firefox Toolbar [20], SiteDigger [47], JUnit [27]

Table 2.1: Some examples of security testing tools. This table is inspired from Curphey et al. [63].

Tool type (category)	SDLC phase
Source-code analyzers	Development, Testing, Predeployment
Web application (black-box) scanners	Testing, Predeployment, Postdeployment
Database scanners	Testing, Predeployment, Postdeployment
Binary analysis tools	Testing, Predeployment, Postdeployment
Runtime analysis tools	Development, Testing, Predeployment, Postdeployment
Configuration analysis tools	Development, Testing, Predeployment, Postdeployment
Proxy tools	Testing, Predeployment, Postdeployment
Miscellaneous tools	Development, Predeployment, Postdeployment

Table 2.2: This table shows the same tool categories as in Table 2.1 along with the SDLC phase in which they are typically applied. This table is inspired from Curphey et al. [63].

## 2.5 Acunetix Web Vulnerability Scanner (WVS)

Acunetix Web Vulnerability Scanner [2] is an automated Web application security testing tool. It can be used to audit a Web application by checking for vulnerabilities such as SQL injection, cross site scripting and many other exploitable vulnerabilities. Additionally, it offers a solution for analyzing off-the-shelf and custom Web applications [54]. It also allow testers to create user defined vulnerability tests that can be added to the existing “library” of vulnerability tests. The tool also allows users to create customized scan profiles in order to perform specific security tests and thereby reduce the total scan time. These scan profiles are sometimes referred to as scan policies [100, 83].

The following six points briefly explain how automated security scanning in Acunetix WVS works [54]:

1. The crawler crawls the entire website by following all the links on the site. Then it displays a tree structure of the website and detailed information of each discovered file (see Figure 2.16).
2. After the crawling process, Acunetix WVS launches vulnerability attacks on each page found, and thereby emulating a hacker.
3. If the port scanner option is enabled, Acunetix WVS will perform network security checks against the services running on the open ports.
4. Acunetix WVS displays each vulnerability as they are detected and places them under an alert node. Alert nodes can either be high, medium or low. It is further possible to look deeper into one vulnerability and look at information like the HTTP response, the source code line and its vulnerable part, stack trace etc. For each discovered vulnerability, Acunetix WVS gives a recommendation on how to fix it.
5. Open ports will be listed along with the security tests that were performed.
6. Finally, it is possible to save a complete scan for later analysis, comparison, or report generation.

The following points give an overview of Acunetix WVS and its main tools and features:

- **Web Scanner:** The Web Scanner is the most important component. It launches the security scan of a Web application which is executed in two steps:
  - First, it uses the crawler function to analyze and build a site structure of the website.

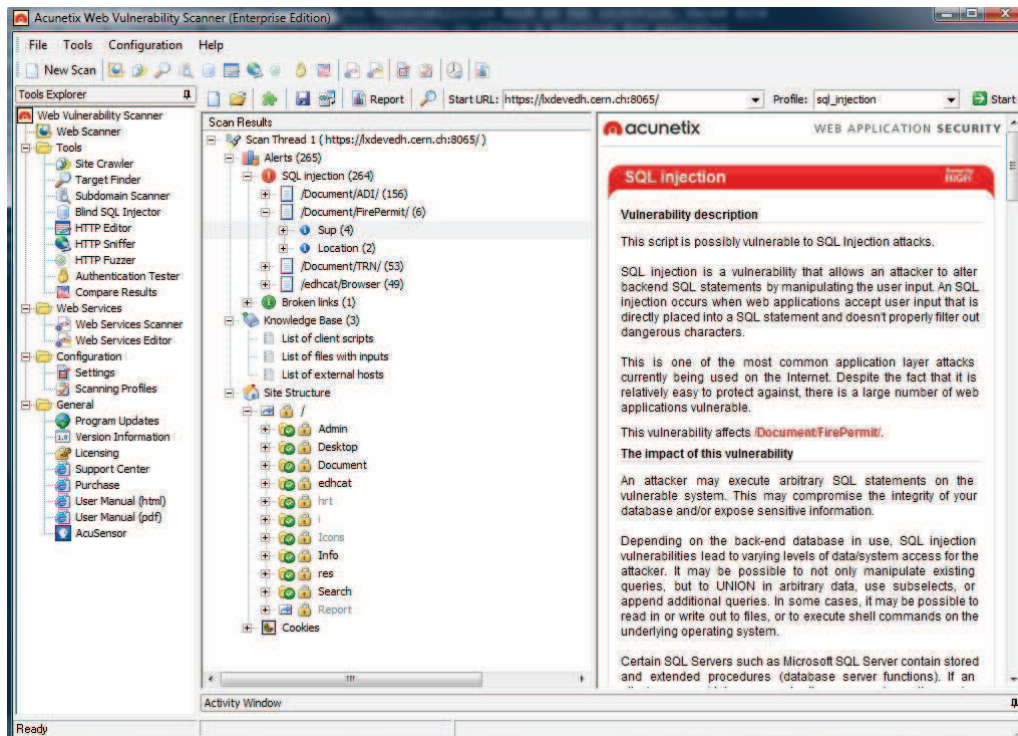


Figure 2.16: This figure shows a screenshot of Acunetix WVS in action. In this example, the scanning profile was set to SQL injection, which means that Acunetix WVS only scans for SQL injections. By further expanding the SQL injection node a list of files containing this vulnerability is shown. Further, expanding the file node reveals the parameters in the file that are leading to SQL injections.

- Then, it launches a series of attacks against the crawled site structure.

The results of a scan are displayed in an Alert Node tree with details on all the vulnerabilities found within the website (see Figure 2.16).

- **AcuSensor Technology:** The AcuSensor Technology leads to the discovery of more vulnerabilities and the generation of less false-positives than a traditional Web Application Scanner [55, 54]. Additionally, it indicates exactly where in the source code the vulnerability lies. This accuracy is achieved by combining black-box scanning techniques with dynamic code analyzes while the source code is executed. Figure 2.17 illustrates how the AcuSensor Technology functions with the rest of Acunetix WVS.
- **Port Scanner and network alerts:** The Port Scanner performs a port scan against the Web server that's hosting the website. If any open ports are found, Acunetix WVS will perform network level security checks

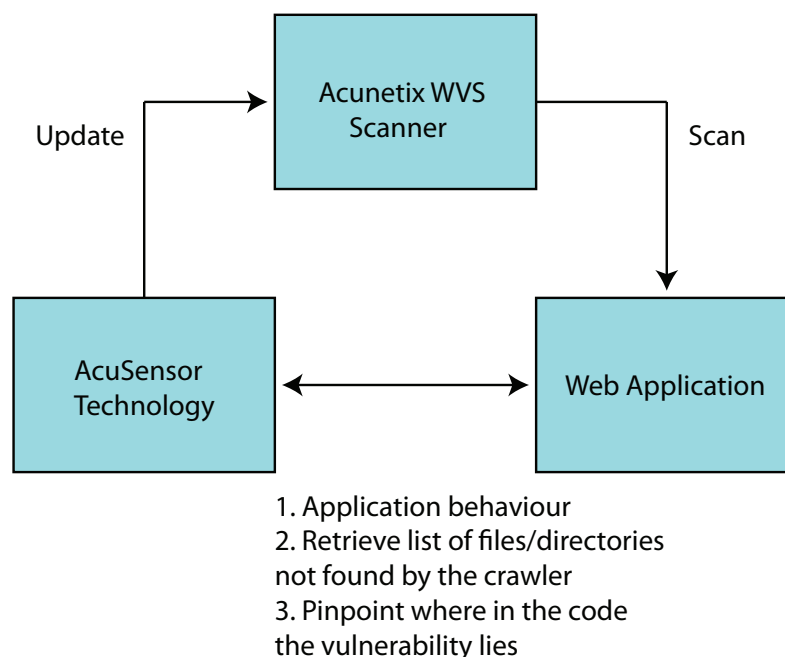


Figure 2.17: An illustration of how the AcuSensor Technology functions with the rest of Acunetix WVS.

against the open ports, such as DNS open recursion tests, badly configured proxy server tests and weak SNMP community strings.

- **Target Finder:** The Target Finder is a port scanner that locates open Web server ports within a given range of IP addresses.
- **Subdomain Scanner:** The Subdomain Scanner identifies active subdomains in a DNS zone.
- **Blind SQL Injector:** The Blind SQL Injector is an automated database data extraction tool where it is possible to perform manual SQL injection tests. The tool is also able to enumerate databases, tables, dump data and also read specific files on the file system of the Web server to check whether a SQL injection is found.
- **HTTP Editor:** The HTTP Editor allows a user to create custom HTTP requests, and debug HTTP requests and responses. Additionally, it includes a tool for encoding and decoding text and URLs to MD5 hashes, UTF-7 formats, etc.
- **HTTP Sniffer:** The HTTP Sniffer acts as a proxy that allows a user to

capture, examine and modify HTTP traffic between a client and a Web server. This tool can be used to:

- Analyze how Session IDs are stored and how inputs are sent to the server.
  - Alter any HTTP request before it gets sent to the Web server.
  - Navigate through parts of the website which cannot be crawled automatically (e.g. because of certain JavaScript code and import results to the scanner).
- **HTTP Fuzzer:** The HTTP Fuzzer allows a user to perform sophisticated buffer overflow and input validation tests. As an example, let's say we have the following URL: `http://www.your-webshop.com/listproducts.php?category=1`. Then, by using the HTTP Fuzzer, it is possible to create a rule to automatically replace the last part of the URL with numbers between 1 and 999. From these generated URLs, only valid results will be reported. This process reduces the amount of manual input significantly.
  - **Authentication Tester:** The Authentication Tester can be used to perform a dictionary attack on login pages which use HTTP authentication or HTML form authentication. This tool uses two text files that contain common usernames and passwords which can be expanded.
  - **Web Services Scanner:** The Web Services Scanner allows a user to scan Web services for vulnerabilities in an automated way.
  - **Web Services Editor:** The Web Services Editor allows a user to import an online or local WSDL for an in depth analyses of WSDL requests and responses by custom editing and execution of Web service operations over different port types. Users can also customize their own manual attacks.
  - **Vulnerability Editor:** The Vulnerability Editor (see Figure 2.18) contains all the security tests that Acunetix WVS launches against Web applications. It allows a user to view, modify and add security tests.
  - **Reporter:** The Reporter allows a user to create reports of scan results. The reports can be created in different level of details depending on whom the report is meant for (e.g. executive summary, developers report, detailed scan report).

### 2.5.1 Why Acunetix WVS?

Acunetix Web Vulnerability Scanner has been selected as the security testing tool for the security tests in this thesis. There are several reasons to this:

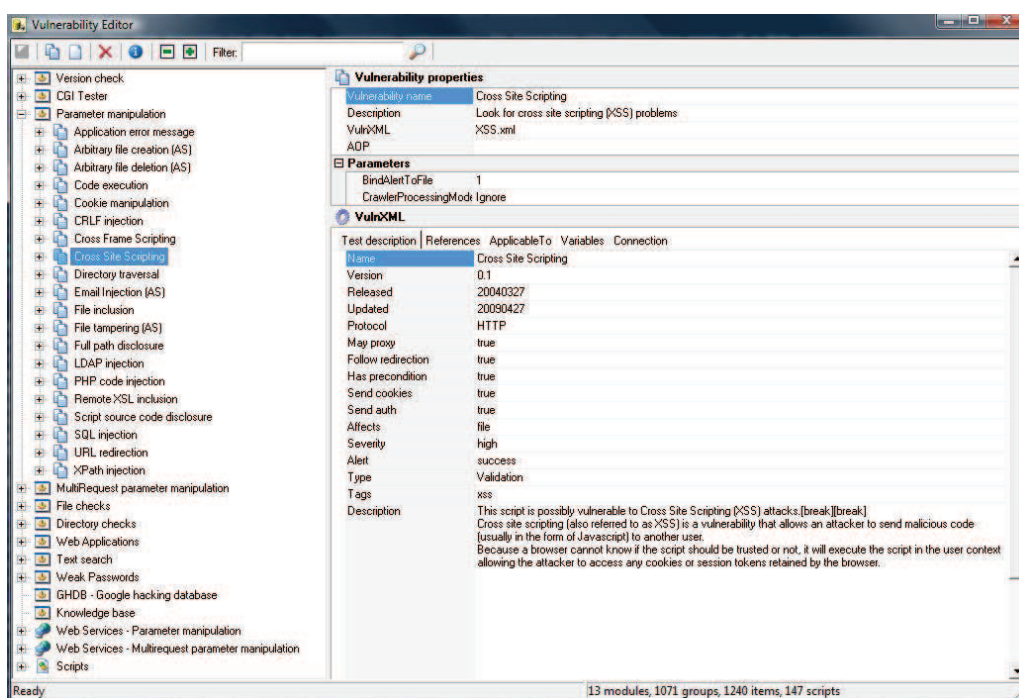


Figure 2.18: This screenshot shows the Acunetix WVS Vulnerability Editor. A user can view, modify or add a security test through the Vulnerability Editor. It is also possible to edit the description of a vulnerability class (e.g. Cross Site Scripting) which appears in a report.

- Acunetix WVS covers a high level of automated security testing coverage.
- Acunetix WVS provides the ability to save crawl results and scan results separately. This may further reduce the total time spent on security testing, which is in line with the overall goal of reducing the time spent on security testing, as defined in point 2 of Section 1.2.
- Acunetix WVS produce less false-positives, which is in line with the overall goal of mitigating false-positives, as defined in point 2 of section 1.2. An independent Web vulnerability scanner comparison done by Ananta Security [55] backs this up. Acunetix WVS, IBM Rational AppScan and HP WebInspect were compared in this comparison. Furthermore, the comparison reveals that Acunetix WVS found 75% of the tested vulnerabilities, while AppScan and WebInspect found 50% and 44% respectively.
- Acunetix WVS is the cheapest Web vulnerability scanner among its commercial competitors (see Table 2.3).



## 2.6 Relevant competitors to Acunetix WVS

There are many security testing tools similar to Acunetix WVS that qualify as relevant competitors:

- **N-Stalker Scanner** is a commercial Web application security testing tool. In addition to scanning for security vulnerabilities in Web applications, it is also built to provide a better control over the Web Application Development Life-cycle [100]. This is done by letting the users create specific security scan policies to cover; (1) development & QA profiles, (2) infrastructure & deployment profiles and (3) penetration testing and security auditing profiles. This tool has a high level of automated security testing coverage, and has the capability of saving scan results and crawl results separately (see Table 2.3).
- **IBM Rational AppScan** [24] is a commercial Web application security testing tool. As N-Stalker Scanner, this tool also let users create customized scanning profiles in order to get better control over the Web Application Development Life-cycle [83]. This tool has a high level of automated security testing coverage, but does not have the capability of saving scan results and crawl results separately.
- **HP WebInspect** [23] is a commercial Web application security testing tool. It is possible to create customized scanning profiles with this tool. Additionally, it is capable of performing security tests during the crawling process (simultaneous crawl and audit) [76], which is different from other similar tools. This tool has a high level of automated security testing coverage, and has the capability of saving scan results and crawl results separately.
- **Cenzic Hailstorm Professional** [9] is a commercial Web application security testing tool. Like the abovementioned tools, it allows users to create customized scanning profiles. This tool has a high level of automated security testing coverage, and has the capability of saving scan results and crawl results separately.
- **OWASP WebScarab** is a free Web application security testing tool. This tool mostly relies on user input (e.g. writing test scripts, editing HTTP requests, etc.) in order to perform specific security tests. It is primarily designed to be used by people who can write code themselves, or at least have a pretty good understanding of the HTTP protocol [37]. It has a medium level of automated security testing coverage and don't have the capability of saving scan results and crawl results separately.

- **Burp Suite** [8] is a free Web application security testing tool. As OWASP WebScarab, this tool is also mostly relying on user input in order to perform specific security tests. It is also primarily designed to be used by people who can write code. This tool has many of WebScarab's functionalities, but has a low level of automated security testing coverage. Additionally, it doesn't have the capability of saving scan results and crawl results separately.

Table 2.3 shows the comparison of Acunetix WVS against the abovementioned tools. The stars in column "Level of automated security testing coverage" represents the broadness of automated security tests the tool covers. Three stars means high coverage, two stars means medium coverage and one star means low coverage. Furthermore, the high, medium and low coverage are based on the following criteria:

- **High coverage** - The tool covers the following automated security check topics:
  - Custom design errors (e.g. XSS, SQL injection and file and directory traversal attacks).
  - Web server exposure (e.g. Web server version vulnerabilities, HTTP protocol vulnerabilities and SSL encryption vulnerabilities).
  - Web signature attacks (e.g. PHP, ASP and J2EE Web application security tests).
  - Cookie exposure checks (e.g. manipulation of cookie information and sensitive information leakage in cookie information).
  - File and directory exposure checks (e.g. search for backup files, configuration files and password files).

Additionally, the tool has to let a user to manually create and perform security tests.

- **Medium coverage** - The tool covers:
  - At least two of the automated security check topics mentioned under "high coverage".

Additionally, the tool has to let a user to manually create and perform security tests.

- **Low coverage** - The tool has the necessary automated functions to retrieve a site structure, and lets the user to manually create and perform security tests.

Site crawling can take several hours for large websites. Tools that make it possible to save the crawl result and the scan result separately, along with providing the ability to create different scanning profiles, can reduce the total time spent on security testing. By saving a crawl result, one can perform many automated security tests on the same crawl result. However, it is important to repeat the crawling process if the site structure has changed.

Security testing tool	Level of automated security testing coverage	Save scan and crawl results separately	Freeware	Commercial	Price (USD)
Acunetix WVS	★★★★	✓	✓ <sup>†</sup>	✓	4'195 \$
N-Stalker Scanner	★★★★	✓	✗	✓	4'899 \$
IBM Rational AppScan	★★★★	✗	✗	✓	7'600 \$
HP WebInspect	★★★★	✓	✗	✓	Starts at 15'000 \$
Genzic Hailstorm Professional	★★★★	✓	✓ <sup>†</sup>	✓	Starts at 35'000 \$
OWASP WebScarab	★★★	✗	✓	✗	Free
Burp Suite	★	✗	✓	✗	Free

Table 2.3: Comparison table of Acunetix WVS against similar security testing tools. <sup>†</sup>Limited to XSS testing only.

## 2.7 Comparison, evaluation and selection

This section describes the criteria that are used for the selection of a security testing methodology for Web applications. The comparison and evaluation scope consist of the security testing methodologies described in Section 2.3. These are Agile Security Testing, the Penetration Testing Approach, and The Open Web Application Security Project (OWASP) Testing Framework.

As mentioned in Section 1.1, the field of security testing (and thereby security testing of Web applications) has yet to be matured; there is still need for security testing methodologies, techniques, and tools. Because of this immaturity of the field, there is a lack of empirical evaluations of security testing methodologies for Web applications. The author's selection criteria are therefore to some extent subjective. The following points describe the criteria that the author find important, and that are used in Table 2.4:

**C.01** - Reducing complexity: Does the security testing methodology reduce the complexity of the testing process?

**C.02** - Increasing efficiency: Does the security testing methodology increase the efficiency, regarding time spent, of the testing process?

**C.03** - Mitigating false-positives: Does the security testing methodology have a phase in order to mitigate false-positives during the testing process?

**C.04** - Increasing knowledge: Does the security testing methodology contribute to increase the security-specific knowledge of the security testers (i.e. the security testing participants) during the testing process?

**C.05** - Postmortem evaluations: Does the security testing methodology have a phase that initiates the security testers (i.e. the security testing participants) to reflect over the vulnerabilities, the development process and the security testing process?

**C.06** - Repository of knowledge: Does the security testing methodology have a phase that initiates the security testers (i.e. the security testing participants) to document and archive *why* certain security decision were made?

Table 2.4 shows which of the security testing methodologies described in Section 2.3 fulfils the abovementioned criteria.

### 2.7.1 Selection justification

The following points explain why the security testing methodologies in Table 2.4 fulfill some criteria while not fulfilling others.

- **Agile Security Testing:**

- Fulfills criterion C.01 by firstly, considering the most necessary points in order to perform a security test on Web applications. There are three main phases that needs to be carried out compared to five and thirteen phases in the Penetration Testing Approach and the OWASP Testing Framework respectively. Secondly, the three main phases require little intervention by security experts, although Agile Security Testing doesn't completely eliminate this need.
- Fulfills criterion C.02 by providing phases that are easy for people with little security knowledge to adapt and apply. This in turn will increase the efficiency, regarding time spent, of the security testing process.
- Does not have a phase for mitigating false-positives. Hence, it does not fulfill criterion C.03.
- Fulfills criterion C.04 by using misuse cases to elicit security requirements, which further gives an overview of potential vulnerabilities. Since the use case notation is known to developers in general, and since the use case notation and the misuse case notation are quite similar (see Figure 2.6), security knowledge is brought closer to developers via misuse cases. Additionally, this helps to decrease the knowledge gap between security experts and software developers.
- Does not have a phase for postmortem evaluations. Hence, it does not fulfill criterion C.05.
- Does not have a phase for documenting and archiving security decisions. Hence, it does not fulfill criterion C.06.

- **A Penetration Testing Approach:**

- Has two phases that relies mostly on security experts' knowledge and thorough planning. These are phase 1 (create a threat model) and phase 2 (build a test plan), which further makes the testing process more complicated. Therefore, it does not fulfill criterion C.01.
- The two phases mentioned in the previous point generally require much time, which further decrease the efficiency, regarding time spent, of the testing process. Hence, it does not fulfill criterion C.02.

- Does not have a phase for mitigating false-positives. Hence, it does not fulfill criterion C.03.
- Fulfills criterion C.04 through the execution of test cases and creation of the problem report. By letting the testers (which might also be developers) create, execute and analyze various security test types (dependency, user interface, design, and implementation testing) they gain specific knowledge of how different vulnerabilities might be exploited and thereby gaining security specific knowledge. Furthermore, the problem reports have to contain reproduction phases, severity and exploit scenarios which are an invaluable resource for distributing security specific knowledge.
- Fulfills criterion C.05 by having a postmortem evaluation phase at the end of each security testing iteration. The postmortem evaluation phase enables the testers to focus on why vulnerabilities were missed during development, and how to improve the development and testing process to prevent given security issues.
- The fact that a postmortem evaluation phase is present, and that one of the goals in this phase is to create countermeasures for improving the development and testing process to prevent security issues, indicates that some documentation must be created for later use. This will further lead to a repository of security specific knowledge over time. Hence, it fulfills criterion C.06.

- **OWASP Testing Framework:**

- Does not fulfill criterion C.01 by firstly, having many phases that relies mostly on security experts' knowledge (phase 1a, 2a, 2b, 2c, 2d, 3b and 5a given in Section 2.3.3) and secondly, by having to many phases. This makes the overall testing process complicated. Although the OWASP Testing Framework is created for a general SDLC which makes it possible to pick and use the necessary phases, they are sometimes closely coupled. E.g. to only carry out a code walkthrough (phase 3a) without carrying out a code review (phase 3b) afterwards would not be of any particular benefit.
- Has thirteen phases that are quite large which together will require much time. Hence, it does not fulfill criterion C.02.
- Does not have a phase for mitigating false-positives. Hence, it does not fulfill criterion C.03.
- Fulfills criterion C.04 through the various reviewing phases (phase 1a, 2a, 2b, 2c, 2d, 3b, 4a and 5a). Although most of these phases

relies mostly on security experts' knowledge, it is possible to apply various activities within these phases in order to reduce the need for security experts. For example; misuse cases [110, 105] can be used in phase 2a, the goal-driven and vulnerability-driven security inspections suggested by SHIELDS [71, 72] can be used in phase 2a, 2b, 2c, and 2d, and the security code review technique suggested by Howard [81] can be used in phase 3b.

- Does not suggest a specific postmortem evaluation phase, but has three phases (5a, 5b, and 5c) in which the testers can continuously get the status about the existing security level. Furthermore, this gives the testers an opportunity to reflect over the vulnerabilities and the security testing process. Hence, it fulfills criterion C.05.
- Does not have a phase for documenting and archiving security decisions. Hence, it does not fulfill criterion C.06.

Based on the abovementioned points, the author have selected Agile Security Testing as the most adequate security testing methodology for Web applications, for the purpose and scope of this thesis. Agile Security Testing mitigates the three main factors that make security testing of Web application more complicated (see Section 1.1):

- By fulfilling criterion C.01, Agile Security Testing mitigates the overall complexity of the testing process. The developers can therefore perform security testing within the short development timeframe (short time-to-market), which is typical for Web applications.
- By fulfilling criterion C.02, Agile Security Testing mitigates the “too time-consuming” attitude against security testing of Web applications.
- By fulfilling criterion C.01, C.02 and C.04, Agile Security Testing shows that security testing of Web applications doesn't lack a significant payoff, which is sometimes how it is considered.



Security testing methodology	Criteria						
	Reducing complexity	Increasing efficiency	Mitigating false-positives	Increasing knowledge	Postmortem evaluations	Repository of knowledge	
Agile Security Testing	✓	✓	✗	✓	✗	✗	
A Penetration Testing Approach	✗	✗	✗	✓	✓	✓	
OWASP Testing Framework	✗	✗	✗	✓	✓	✗	

Table 2.4: Comparison table of security testing methodologies for Web applications.



# Chapter 3

## Current situation

This chapter gives a short introduction to CERN and the AIS group at CERN. Furthermore, it explains how security testing is currently done in the group, and explains why there is a need for a security testing methodology in the group. Finally, it shows a high level risk analysis of the AIS group's Web applications. The risk analysis is carried out by using the CORAS security risk analysis methodology.

### 3.1 CERN - The European Organization for Nuclear Research

CERN is the world's largest particle physics laboratory. It is located in the outskirts of Geneva, Switzerland and was founded the 29<sup>th</sup> of September, 1954, by twelve European states. It currently has twenty European Member States<sup>1</sup> and eight Observer States and Organizations<sup>2</sup>. CERN's mission is clearly stated in Article two in the convention that established CERN in 1954 [15]:

*“The Organization shall provide for collaboration among European States in nuclear research of a pure scientific and fundamental character, and in research essentially related thereto. The Organization shall have no concern with work for military requirements and the results of its experimental and theoretical work shall be published or otherwise made generally available.”*

---

<sup>1</sup>The Member States are Austria, Belgium, Bulgaria, the Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Italy, the Netherlands, Norway, Poland, Portugal, the Slovak Republic, Spain, Sweden, Switzerland and the United Kingdom.

<sup>2</sup>The Observer States and Organizations are the European Commission, India, Israel, Japan, the Russian Federation, Turkey, UNESCO and the USA.

This statement not only triggered the concept of open international scientific cooperation, but also gave CERN an important role in the driving forces that made the political boundaries between the west and east more transparent. This was particularly important in a time when the world was recovering after the Second World War.

CERN has about 2500 fulltime employees, and its facilities are used by approximately 8000 scientists, representing 500 universities and over 85 nationalities [1]. Figure 3.1 shows the organizational structure of CERN.

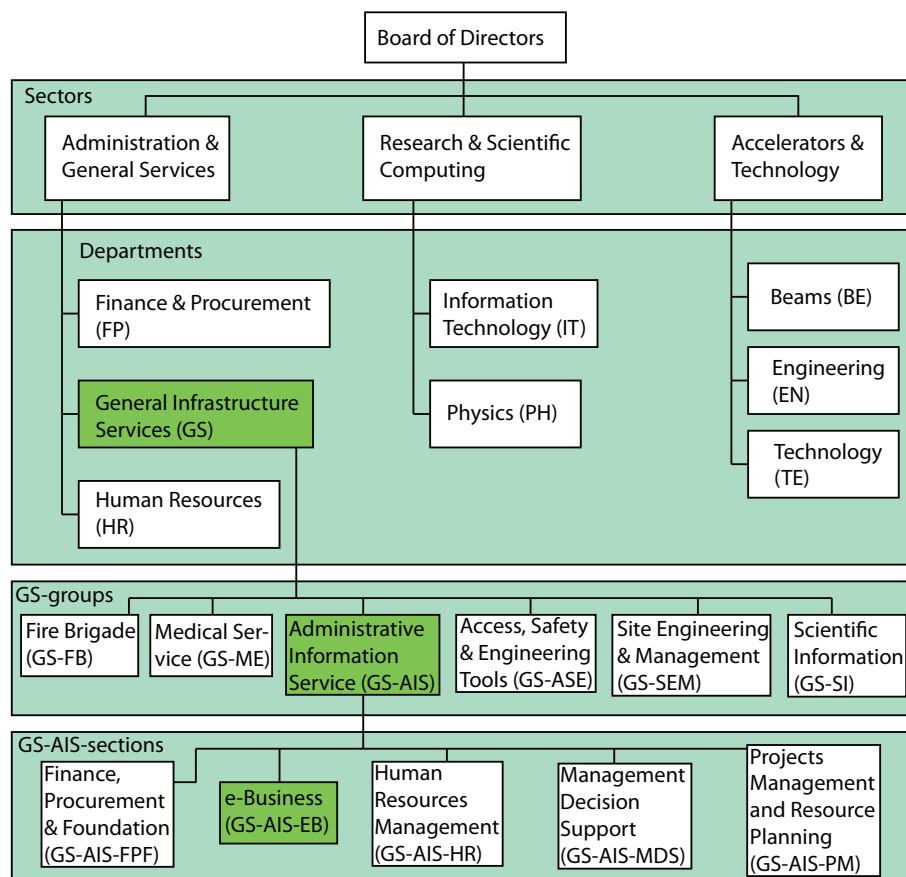


Figure 3.1: The organizational structure of CERN. Only one department (GS), and one group (GS-AIS) is expanded for illustration.

CERN provides physicists with the necessary tools (accelerators and detectors) in order to carry out fundamental research. One of the things CERN is mostly known for is the Large Hadron Collider (LHC). The LHC is the world's largest and highest-energy particle accelerator, intended to collide opposing particle beams at nearly the speed of light. The main purpose of LHC and its experiments is to test and reveal whether some predictions of high-energy physics

really are true or not (e.g. the Higgs boson). Figure 3.2 shows a map of LHC and an overall view of the LHC experiments.

CERN also plays a vital role in developing cutting edge technology. Some of the most important contributions in this respect are the following:

- The World Wide Web [52]
- Cancer therapy
- Medical and industrial imaging
- Radiation processing
- Electronics
- Measuring instruments
- New manufacturing processes and materials

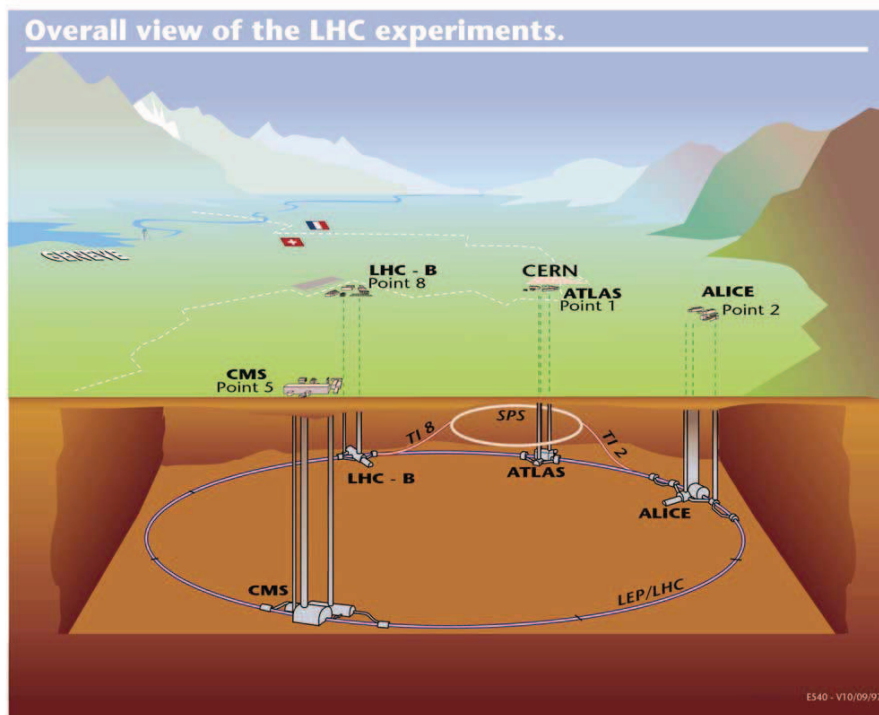


Figure 3.2: Map of LHC and an overall view of its experiments. The big circle in the figure illustrates the 27 kilometres long LHC tunnel. At the deepest, it is 175 meters beneath the earth.

### 3.1.1 The Administrative Information Services (AIS) group

The AIS group of the GS department (see Figure 3.1) has the responsibility for all administrative applications and corporate data at CERN. The main objective is to provide CERN with a set of integrated and reliable corporate Information Systems. In addition, they design and develop software to cover areas such as Workflow, Electronic Document Processing, Budget Management and Project Management where the current market offer does not match CERN requirements. The following services are covered by the group [51]:

- Analyse and specify the functional requirements of administrative applications
- Purchase, develop, implement, and maintain administrative applications and all tools required for optimal exploitation of the corporate information
- Support users of these applications, prepare user documentation and provide training

Furthermore, the business areas that the group is responsible for are the following [51]:

- Workflow/Self service (EDH)
- Management Reporting and Simulation
- Project Management and Contract Management
- Finance
- Purchasing and related Logistics (movement of goods on site, inbound, and outbound)
- Foundation which contains all reference information
- Human Resources Management, E-recruitment and Payroll
- Material Management & Logistics (Replenishment, Warehouse Management, E-catalog, etc.)
- Import/Export (supporting all Import, Export and Expedition procedures)

In order to provide a unique, coherent and integrated environment for all applications, tools, and documentation, AIS makes virtually all functionality available through a Web interface (i.e. through Web applications). This is another example of how important Web applications are for modern information interaction. The AIS group has currently six main applications in operation for users

inside and outside CERN. One of these applications is the Electronic Document Handler (EDH), which has approximately 11'000 users worldwide. Furthermore, it is CERN's largest administrative Web application.

Having the responsibility for developing and maintaining the Web applications that are vital for CERN's various business areas (given in the abovementioned points), and because these various business areas will naturally produce many requests for updating and modifying the applications whenever a change in a given business area is made, the AIS group have to use a development methodology that can produce results efficiently. The AIS group therefore use Scrum (described in Section 3.2) as their main development methodology. The Scrum methodology fits the AIS group's development process by providing maximum flexibility and appropriate control of the development process. These are important points that should be present when developing Web applications (see Section 1.1).

## 3.2 The Scrum methodology: A short introduction

The Scrum methodology was developed because the traditional SDLCs were regarded as being too complicated and complex, even the phases and phase processes in agile development methodologies such as the Spiral methodology and the Iterative methodology were regarded as being linear like the well known Waterfall model [108]. The Scrum methodology intends to enable development teams to operate adaptively within a complex environment using imprecise processes. It does this by assuming that the analysis, design, and development processes in the Sprint phase (see Figure 3.3) are unpredictable. This is also the key difference between the Scrum methodology and other development methodologies. The Scrum methodology has the following three main phases, along with their respective sub-phases [108]:

- **Pregame**
  - *Planning*: The definition of a new release is made in this phase based on the existing backlog. Additionally, an estimation of its schedule and cost is made. In case a new system is to be developed, the system concept is to be defined and analyzed in this phase, otherwise limited analysis is carried out.
  - *System Architecture/High Level Design*: In addition to system architecture modification and high level design, this phase also contains the process of designing how the backlog items will be implemented.

- **Game**

- *Sprints*: The sprint phases act as the engine in which the product backlog items are developed. The goal is to develop new release functionality within a period of one to four weeks. During a sprint, the variables of time, requirements, quality, cost and competition are always regarded through the Scrum meetings. Depending on these variables, a backlog item in a sprint may be discarded, set on pause, expanded, etc. Therefore, there is no guarantee that new release functionality will be 100% ready for production. Hence, the right-most item in Figure 3.3: “Potential Shippable Product Increment”. The complete system is developed by carrying out multiple, iterative sprints. One sprint consists of the four below-mentioned phases.

- \* *Develop*: This phase consists of either modifying a backlog item, or developing a new backlog item. Either way, the following activities of the new/modified backlog item are carried out; domain analysis, design, development, implementation, testing and documentation.

- \* *Wrap*: An executable version of the result in the previous phase is made in this phase.

- \* *Review*: In this phase, the team has a meeting where they present the resulting work of the sprint, and for reviewing the progress. This phase also allows the team to discuss and resolve issues, which may further lead to new backlog items. Additionally, potential risks are reviewed and appropriate responses defined.

- \* *Adjust*: If the result from the Review phase leads to changes, then these are defined as product backlog items and added to the product backlog in this phase.

- **Postgame**

- *Closure*: The closure phase is where the release preparations, and the release itself is made. This includes the following activities: Integration, system test, user documentation, training material preparation, and marketing material preparation.

A full description of the Scrum methodology is beyond the scope of this thesis, and is therefore not described any further. The reader is referred to Scrum Development Process [108], and Agile Software Development with Scrum [109] for a detailed description.



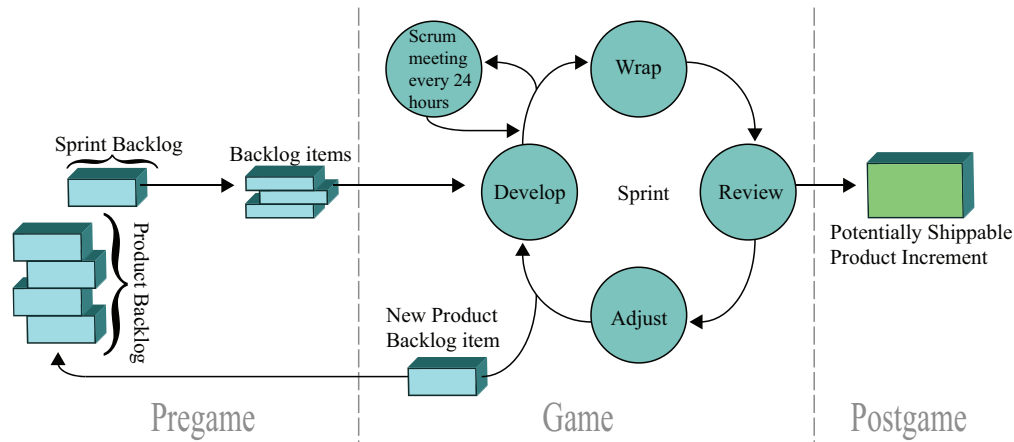


Figure 3.3: *The Scrum SDLC. This figure is inspired from Schwaber [108].*

### 3.3 The need for a security testing methodology

Having the overall responsibility for all administrative applications and corporate data at CERN, the AIS group must make sure that the security is built in their Web applications with respect to confidentiality, integrity and availability (CIA). Some of the information that is handled by the AIS Web applications, and that is of utmost importance to consider are:

- Access Controls
- Medical Data (CERN has its own medical service, as shown in Figure 3.1)
- Administrative Data
- Hostel Data (e.g. credit card information used at the CERN hostels)

Besides developing the applications with CIA in mind, a security testing methodology for Web applications must be in place in order to verify or falsify a given adequate level of security. The security testing at the AIS group is currently done in two main steps (see Figure 3.4):

1. A penetration testing is carried out in the postgame phase. The AIS group does not use any Web Vulnerability Scanners currently, but have guidelines that the testers may use to manually perform penetration tests.
2. The testers create a report of the findings after the penetration testing. The report is used as a basis to create countermeasures for the vulnerabilities. Then, the countermeasures are added in the product backlog. Finally, the vulnerability gets mitigated for the next product increment

by performing the Scrum phases on the particular backlog item containing the vulnerability countermeasures.

Additionally, the penetration tests are often carried out in an ad hoc fashion. I.e. they are not always carried out before each and every “Potentially Shippable Product Increment” as indicated in Figure 3.4.

Apparently, the AIS group is performing penetration testing without considering security testing before and during development. This gives the AIS group very little involvement and control of the security testing process. Furthermore, this approach to security testing is in line with the well known (well known of its criticism and drawbacks) “penetrate and patch” paradigm. There are several disadvantages by performing security testing in this way:

- There are no activities that considers security testing at beforehand (e.g. creation of security requirements). This gives the testing party no basis to use for verifying whether security requirements are fulfilled or not. Without knowing the potential vulnerabilities that the system under test has (which may be discovered via misuse cases) may let undiscovered vulnerabilities to lie dormant for years only to be discovered by malicious users.
- It may seem that this approach is efficient when it comes to the amount of time that is used for the security testing process itself, but the fact is that, by only carrying out penetration testing at the end of the SDLC and thereby fixing the problem, is the second most expensive approach (see Table 1.2). One of the factors to why it is so expensive is that it takes a lot of time to mitigate vulnerabilities at a very late stage. In order to mitigate the vulnerabilities one needs to get educated about the underlying problem, and dig through source code at a stage where it is quite complex. Hence, this approach is not efficient regarding the amount of time spent.
- This approach will give little or no security specific knowledge to developers because it doesn't have activities that bring security specific knowledge closer to developers (e.g. misuse cases).
- The previous points show that this approach is lacking a significant payoff, both economically and educationally.

The abovementioned points indicates that the AIS group needs a new security testing methodology integrated in their development process.

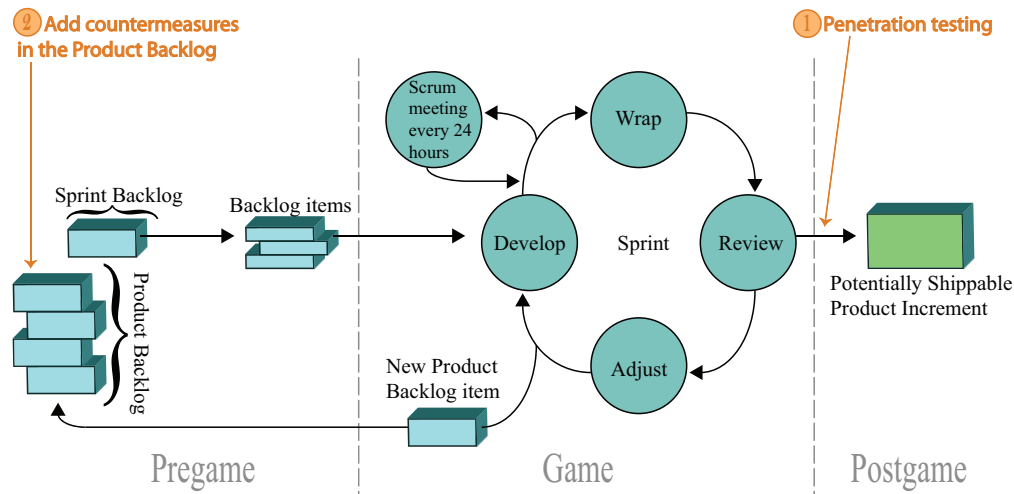


Figure 3.4: The current security testing methodology performed at the AIS group. I.e. penetration testing in the postgame phase, and adding the vulnerability countermeasures to the product backlog after a penetration testing.

## 3.4 Risk analysis

In order to get a high-level understanding of the design flaws, the potential threats, the vulnerabilities, and the associated risks and their consequences that a system might possess, a security risk analysis needs to be conducted. Using the results from a security risk analysis one can further derive countermeasures, conduct cost-benefit analysis, and make critical business decisions. The CORAS methodology is one way of performing security risk analysis.

### 3.4.1 The CORAS methodology: A short introduction

CORAS [111] is a methodology for conducting security risk analysis, specially developed to support structured brainstorming for risk identification, risk estimation and risk treatment [79]. The CORAS methodology consists of seven main steps. The following points briefly describe the steps:

- *Introduction* (step 1): Introductory meeting where the client presents the overall goals of the analysis and the target they wish to have analyzed. The analyst gathers information based on the presentations and the discussions that take place.
- *High level analysis* (step 2): A meeting with the client where the analysts present their understanding of what they learned from the introductory meeting, and from studying the client's relevant documentation. A high-level security analysis is also made in this step. This analysis will be used

to help with directing and scoping the detailed analysis in the following steps.

- *Approval* (step 3): A more detailed description of the target that is to be analyzed, the assumptions and the preconditions are made. This step is terminated when the client has approved this documentation.
- *Risk identification* (step 4): A workshop is executed to identify as many unwanted incidents, threats, vulnerabilities and threat scenarios as possible. The participants are typically people with expertise on the target of the analyses.
- *Risk estimation* (step 5): A workshop is executed to estimate likelihood and consequence values for each of the unwanted incidents discovered in step 4.
- *Risk evaluation* (step 6): A first overall risk picture is given to the client. Typically, this step introduces some adjustments and corrections.
- *Risk treatment* (step 7): Treatment identification, together with a cost/benefit analysis of the treatments, is given to the client. This step is best organized as a workshop.

Furthermore, CORAS provides a customized language which consists of a graphical modeling part and a textual syntax part. The textual language is used to describe the graphical models. But the graphical models are used as the main communication medium for representing the risk analysis. The CORAS graphical modeling language [65] has five different diagrams; asset diagrams (also known as asset overview diagrams), threat diagrams, risk diagrams, treatment diagrams and treatment overview diagrams. The **asset diagrams** are used in step 1 (introduction) to specify the stakeholders of the security analysis and their assets. The **threat diagrams** are used in step 4 (risk identification) to identify and document how vulnerabilities may be exploited by threats, that may further trigger unwanted incidents. In addition, the threat diagrams are used to identify and document which assets the unwanted incidents affect. The threat diagrams are further used as input for step 5 (risk estimation) to create **risk diagrams**. Risk diagrams specify the threats, the risks initiated by the threats and the assets that may be harmed by them. The risk diagrams are further used to present the overall risk picture in step 6 (risk evaluation). The risk representation is then compared to a predefined risk tolerance list. From this comparison, it is possible to decide which risks need treatments. After identifying the risks that need treatments, the result is used as input for step 7 (risk treatment) to create **treatment diagrams**. Treatment diagrams are

used to reason about countermeasures for the identified risks. **Risk overview diagrams** are used to present a high level summary of the findings from an analysis. In particular, it is meant to present the findings in such a way that they are well understood by the decision makers. The graphical diagrams in CORAS are built up of the elements showed in Figure 3.5.

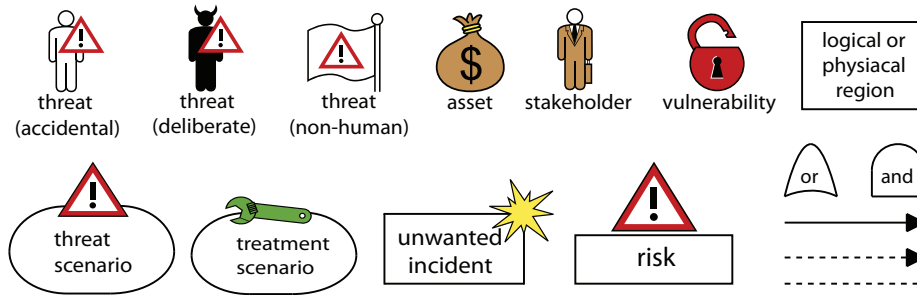


Figure 3.5: *The CORAS graphical modeling notation. The solid arrow is used between threats, vulnerabilities, threat scenarios, unwanted incidents and assets, while the dashed arrow is used for treatments. This figure is adapted from Erdogan and Baadshaug [70].*

### 3.4.2 Conducting the risk analysis

In this section, the CORAS methodology is used to show the potential risks that the Web applications belonging to the AIS group might have. As explained in Section 3.1.1, the AIS group has a collection of different software systems that covers many business areas. Because of this, it would be quite extensive to conduct a risk analysis based on the potential threats for the specific functionalities in their various software systems. In stead, the author has chosen to conduct a risk analysis based on the assets (that are processed by the software systems belonging to the AIS group) that are of importance for the AIS group and those that are using their software systems. The author has chosen to conduct the risk analysis using CORAS, because it defines assets as a starting point. The purpose of this risk analysis is to show the assets that are of importance to the AIS group (and consequently to CERN), the potential threats that may harm the assets, the unwanted incidents (i.e. the risks) that the threats may trigger, and their corresponding risk estimation (likelihood and consequence). The formal processes such as presentation of the CORAS methodology for a client, and the execution of workshops will not be carried out. For this reason, only relevant steps from the CORAS methodology will be used:

1. The focus and scope of the risk analysis will be given.

2. The assets and their corresponding stakeholder will be listed, along with a high-level analysis (risk table).
3. A likelihood scale, a consequence scale, and a risk matrix will be given.
4. A threat diagram containing the risks will be given.
5. Risk estimation will be carried out on the resulting threat diagram from step 4.
6. Based on the risk estimation values from step 5, a risk matrix will be created.
7. Finally a treatment overview diagram showing high level treatments will be given.

**STEP 1** The focus and scope of the risk analysis:

The focus of the risk analysis is data security. This includes:

1. Business sensitive data for CERN that's being handled by the AIS applications.
2. Data that is directly related to CERN employees.

The scope is on data security at CERN processed through the AIS applications.

**STEP 2** Assets, stakeholders and high-level analysis (risk table):

Figure 3.6 shows an asset diagram containing the stakeholders and their related assets, i.e. which asset is important to whom, and whom does it affect directly? Furthermore, there is an indirect asset (CERN's reputation) which is affected by other assets. Table 3.1 shows the assets ranked by importance. Table 3.2 shows a high level risk table describing the unwanted incidents.

Asset	Importance	Asset Type
CERN's reputation	(Scoped out)	Indirect
Salary data	2	Direct
Employee personal data	1	Direct
Medical data	1	Direct
Credit card information (Hostel)	1	Direct
Access controls	1	Direct
Inventory data	3	Direct

Table 3.1: This asset table shows the importance of an asset denoted by a value from 1 to 3. 1 = very important, 2 = important, 3 = minor important.

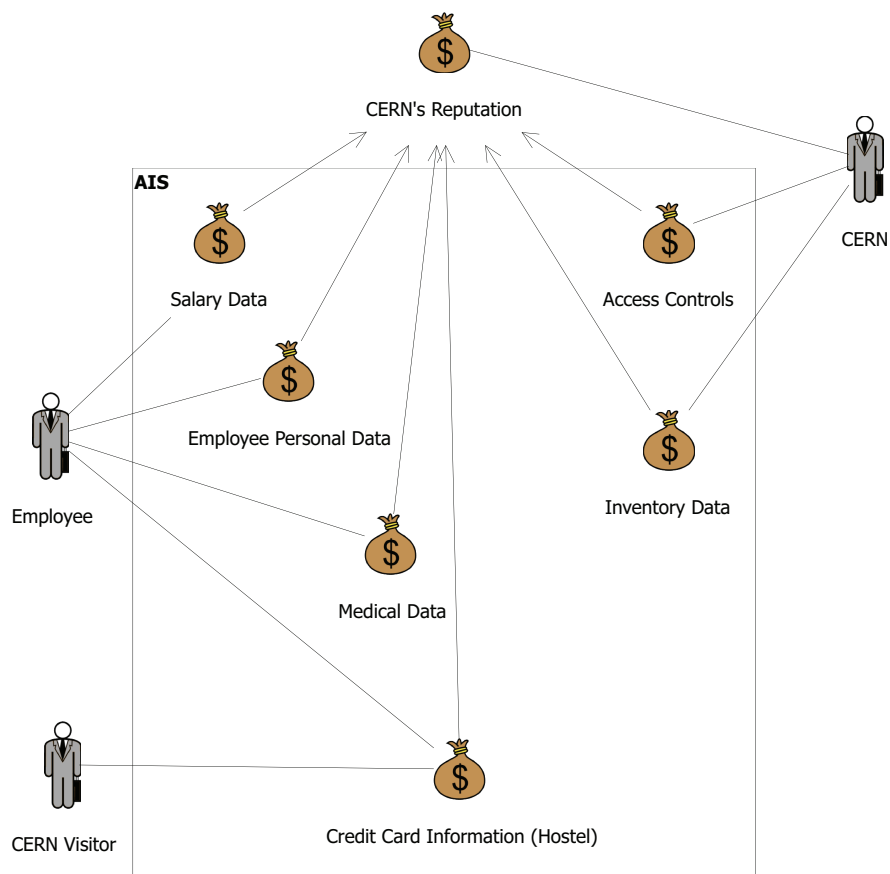


Figure 3.6: This asset diagram shows the stakeholders and their related assets. This asset diagram was created using CORAS editor v.2.0.b5 [16].


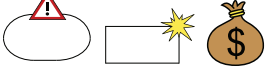

		
Who/what is the cause?	How? What may happen? What does it harm?	What makes this possible?
Employee	Responds to falsely forged e-mail giving away credentials, which further let malicious users gain access to the system and thereby access to the particular employee's personal data	Lack of security training/awareness among the employees
Employee (Software Developer)	Software developers don't use software security practices during development, which may further let undiscovered vulnerabilities lay dormant in the applications	Lack of software security development practices in the SDLC
Malicious Employee (Insider)	Misuse his/her position to get access to the various assets shown in the AIS frame in Figure 3.6	Weak information restriction rules (least privilege)
Malicious Employee (Insider)	Logs into the AIS system using his/her credentials and thereby tries to obtain information by performing various attacks such as SQL injection and Cross Site Scripting	Weak or none input validation on the client and the server side
Hacker	Logs into the AIS system by brute force attacking the login page (e.g. dictionary attack), and thereby gets access	Weak username/password combinations
Hacker	Gets access to the AIS system and performs various attacks such as SQL injection and Cross Site Scripting in order to gain further information (assets shown in the AIS frame in Figure 3.6)	Weak or none input validation on the client and the server side

Table 3.2: High level risk table describing the unwanted incidents.



**STEP 3** Likelihood scale, consequence scale, and the risk matrix:

Likelihood scale	Likelihood description
1	Rarely
2	Sometimes
3	Regularly
4	Often

Table 3.3: The likelihood scale used in the risk analysis for the unwanted incidents. This table is inspired from CORAS Tutorial [64].

Consequence scale	Consequence description
1	Harmless
2	Moderate
3	Serious
4	Catastrophic

Table 3.4: The consequence scale used in the risk analysis for the unwanted incidents. This table is inspired from CORAS Tutorial [64].

Likelihood \ Consequence	Rarely	Sometime	Regularly	Often
Harmless				
Moderate				
Serious				
Catastrophic				

Table 3.5: The resulting risk matrix by combining the likelihood table and the consequence table. The risks that fall in the green area are acceptable risks. The risks that fall in the red area are the most critical ones and are those that need urgent countermeasures. This table is inspired from CORAS Tutorial [64]. Generally, risk matrices have values that are neither critical nor acceptable. These values need to be constantly monitored and are placed in a yellow area in a risk matrix. The yellow area has been omitted in this risk matrix, because they too, are regarded as critical in this case.

**STEP 4 and 5** Threat diagram and risk estimation:

In order to limit the scope of the risk analysis, only the most important assets (given in Table 3.1) are considered, and the indirect assets are scoped out as they are outside the AIS domain (see Figure 3.6). Figure 3.7 shows a threat diagram illustrating threats that may trigger unwanted incidents, which may further harm the most important assets. The risk estimations for each unwanted incident are also given in the diagram. The risks that are considered in Figure 3.7 are the following:

**R.01** Personal data is stolen and used for identity theft

**R.02** Personal data is deleted

**R.03** Medical data is modified

**R.04** Credit card information is sold or used

**R.05** Credit card information is removed so the Hostel won't get payed

**R.06** Access information is stolen and used by malicious users to get access to the applications

**R.07** Access information is modified and the victim is denied access to use the applications

**STEP 6** Risk evaluation:

The following table shows which of the risks are acceptable and which of the risks that need urgent treatment. The calculations are derived from the results in Figure 3.7.

	Likelihood	Rarely	Sometime	Regularly	Often
Consequence					
Harmless					
Moderate			R.02, R.07		
Serious		R.05			
Catastrophic			R.03		R.01, R.04, R.06

Table 3.6: This risk evaluation table shows which of the risks in Figure 3.7 are in the “green zone” and which are in the “red zone”. The risks in the “green zone” are acceptable, while those in the “red zone” are unacceptable.

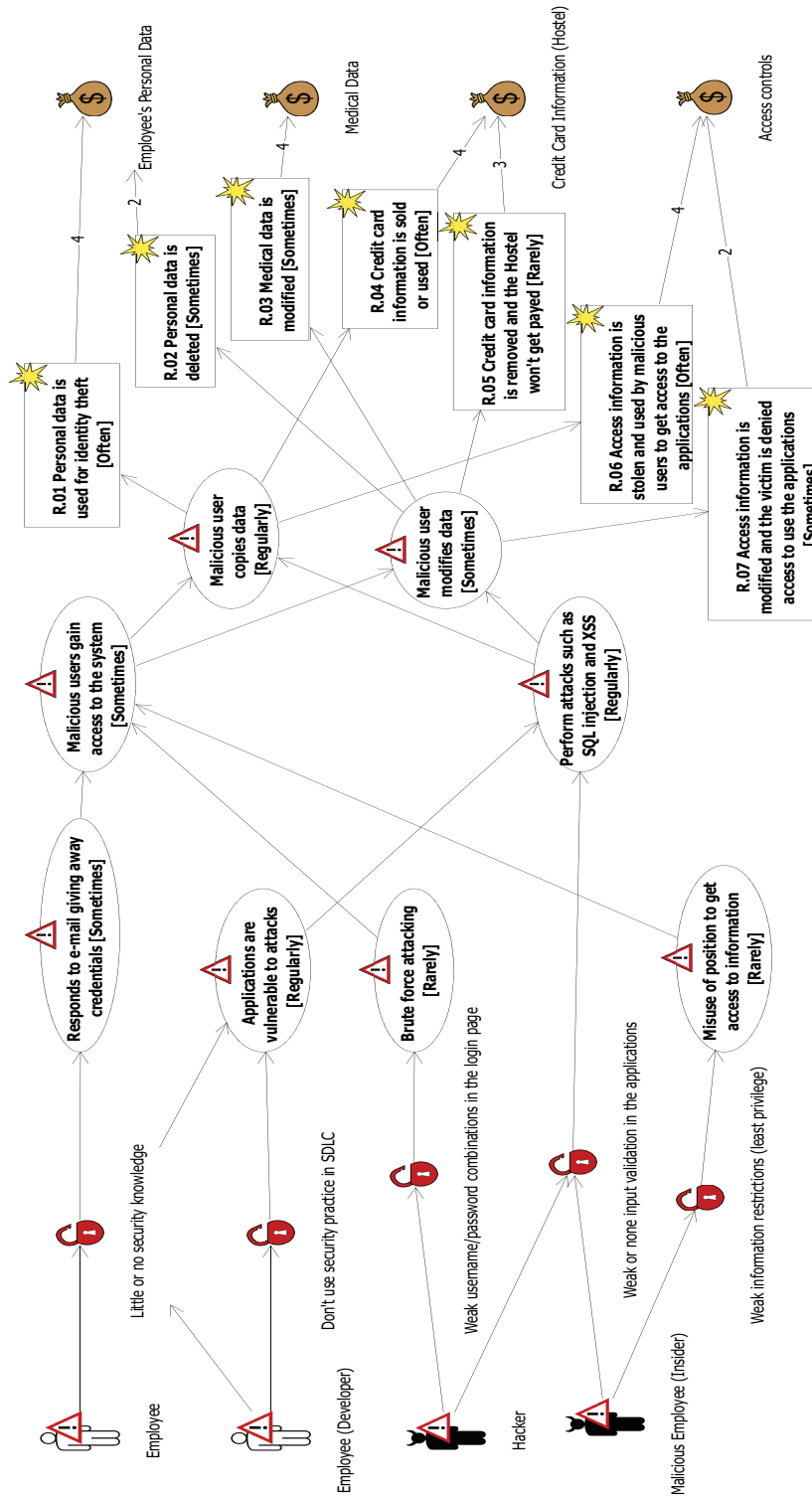


Figure 3.7: Threat diagram and the risk estimations. This threat diagram was created using CORAS editor v.2.0.b5 [16].

**STEP 7** Risk treatment:

Figure 3.8 shows a risk treatment overview diagram. The risks that are shown in the risk treatment diagram (R.01, R.03, R.04 and R.06) are those with the highest risks according to Table 3.6. The logical OR gates in Figure 3.8 are showing that one input element to the OR gate is affecting all the elements that the gates are pointing at. E.g. as shown in Figure 3.7, an employee that has little or no security specific knowledge may trigger potential threats by responding to an e-mail giving away his/her credentials. By following the resulting risks that this may lead to, shows that such an action from an employee may lead to all the risks that are given in the same figure. Hence, the logical OR gate in Figure 3.8 that connects the threats with the risks. This is the same for the logical OR gate that connects the treatments with the threats. E.g. the “use security practices in the SDLC” treatment would mitigate R.01, R.03, R.04 and R.06.

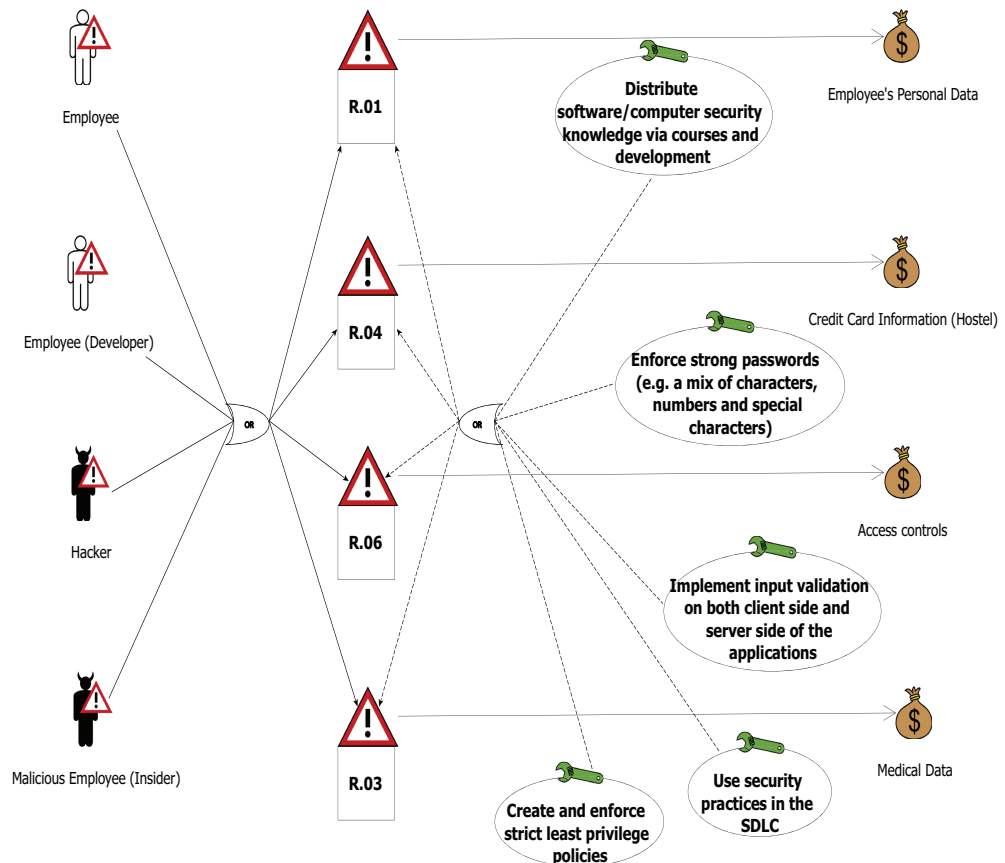


Figure 3.8: This risk treatment overview diagram shows the high level treatments (countermeasures) in order to mitigate the risks. This treatment overview diagram was created using CORAS editor v.2.0.b5 [16].

# Chapter 4

## Contribution

This chapter describes the author's contributions to security testing of Web based applications, which are the following:

1. Extending the Agile Security Testing methodology (that was selected in Section 2.7.1 for the purpose of this thesis) to make it support all the criteria defined in Section 2.7.
2. Integrating the extended Agile Security Testing methodology (at a proof of concept level) in the AIS group's SDLC.
3. Performing security tests on parts of EDH using the extended Agile Security Testing methodology and thereby measuring its efficiency, compared to existing ad hoc ways of performing security tests.

### 4.1 Extending Agile Security Testing

As stated in Section 2.7.1, Agile Security Testing is selected as the most adequate security testing methodology for Web applications, for the purpose and scope of this thesis. Although Agile Security Testing fulfills the criteria that mitigates the three main factors that make security testing of Web applications more complicated (see Section 2.7.1), it does not fulfill all the criteria defined in Section 2.7. For reasons given in Section 2.7.1, Agile Security Testing doesn't fulfill the following criteria:

**C.03** - Mitigating false-positives: Does the security testing methodology have a phase in order to mitigate false-positives during the testing process?

**C.05** - Postmortem evaluations: Does the security testing methodology have

a phase that initiates the security testers (i.e. the security testing participants) to reflect over the vulnerabilities, the development process and the security testing process?

**C.06** - Repository of knowledge: Does the security testing methodology have a phase that initiates the security testers (i.e. the security testing participants) to document and archive *why* certain security decision were made?

In order for Agile Security Testing to support the abovementioned criteria, the author extends Agile Security Testing by adding the following activities. The result (as shown in Figure 4.3) is named Extended Agile Security Testing by the author.

1. False-positives (i.e. nonexistent bugs that are reported as detected by a testing tool) and false-negatives (i.e. existing bugs that are not detected by a testing tool) are known to be a problem in automated software security testing. This problem is especially known to be produced by the static analyzer tools. Furthermore, if static analyzers are configured so that they don't produce any false-negatives (i.e. 100% false-negative proof), then the developers tend to be flooded with false-positives [101]. A high rate of false-positives creates high workload and makes it difficult to find and fix the actual bugs in the software. The author therefore integrates a false-positive mitigation process with the penetration testing process. The penetration testing process is regarded as a part of point 3 in Agile Security Testing (see Section 2.3.1). The false-positive mitigation process is to be carried out in the following way:
  - (a) The penetration testing tool is used to perform a penetration test.
  - (b) After a penetration test, the result is reviewed and the false-positives are marked so that they won't be registered as bugs next time the tool performs a penetration test. Each marked false-positive vulnerability is associated with its respective Web site.

In the abovementioned approach, the penetration testing tool will continuously be taught about the false-positives for a specific application, and thereby continuously mitigate false-positives. However, this approach is dependent on either: (1) the penetration testing tool has the ability to mark and remember specific false-positives or, (2) the penetration testing tool has the ability to import a false-positive repository (e.g. false-positive database, XML file, etc.). Fortunately, Acunetix WVS allows users to mark a specific reported vulnerability as a false-positive (the aforementioned point 1). The specific false-positive is then added to a false-positive

repository and stays there until it is removed. However, the classification of false-positives as security vulnerabilities is not an easy task, and is dependent on the tester's security specific knowledge and experience on the underlying analysis tool. An experiment carried out by Baca et al. [59] reveals that testers who have security specific knowledge *and* experience in using the underlying analysis tool are the best vulnerability detectors in this respect. Figure 4.1 shows how a given vulnerability might be registered as false-positive in Acunetix WVS, and Figure 4.2 shows how marked false-positives are organized in Acunetix WVS. Acunetix WVS saves all registered false-positives in an XML file, which makes it easy to distribute among Acunetix WVS users. This approach introduces a phase in the testing process that mitigates false-positives, and thereby fulfills criterion C.03.

2. In order to continuously harden the security testing process, a postmortem phase needs to be in place at the end of the security testing process. A postmortem phase is a meeting session that's executed by the security testers. It enables them to reflect over the vulnerabilities, the development process and the security testing process. The postmortem phase is realized by executing the following steps:
  - (a) Provide answers to why certain vulnerabilities were missed during development.
  - (b) Improve the issued development process in order to mitigate or isolate the underlying vulnerabilities.
  - (c) Create, or find, or improve a security testing activity in order to detect the underlying vulnerabilities.

Criterion C.05 is fulfilled by adding the abovementioned postmortem activity in Agile Security Testing.

3. According to Rus et al. [106], a software organization's main asset is its intellectual capital. One obvious problem in this respect is that intellectual capital lies within the minds of the employees. As experienced people leave the organization, so does the knowledge with them. In order to keep the knowledge alive within the organization, knowledge management must be present in the organization. Knowledge management is beyond the scope of this thesis, but by adding a phase in Agile Security Testing that initiates the security testers (that may be developers, QA people, decision makers etc.) to document and archive *why* certain security decision were made, security specific knowledge can be kept in repositories, and thereby



Figure 4.1: By first selecting a false-positive vulnerability, and then clicking on “Mark this alert as a false positive” (indicated by the red arrow), a false-positive is marked. The specific false-positive won’t be treated as a vulnerability next time Acunetix WVS encounters it on the respective Web site/application.

kept alive within the organization. This phase is named “Repository of knowledge” by the author, and is realized by executing it during the development and the review phase of the SDLC. Additionally, it is important to maintain the repository for the underlying system while it is in operation. I.e. if vulnerabilities are discovered while the system is in operation, then these have to be documented as well. By doing this, countermeasures can be prioritized and new iterations of the testing process can be initiated. With this, criterion C.06 is fulfilled.

Figure 4.3 shows the Agile Security Testing steps described in Section 2.3.1 on the left hand side of the figure. The Extended Agile Security Testing (hereby EAST) that contains the abovementioned three steps (in addition to the Agile



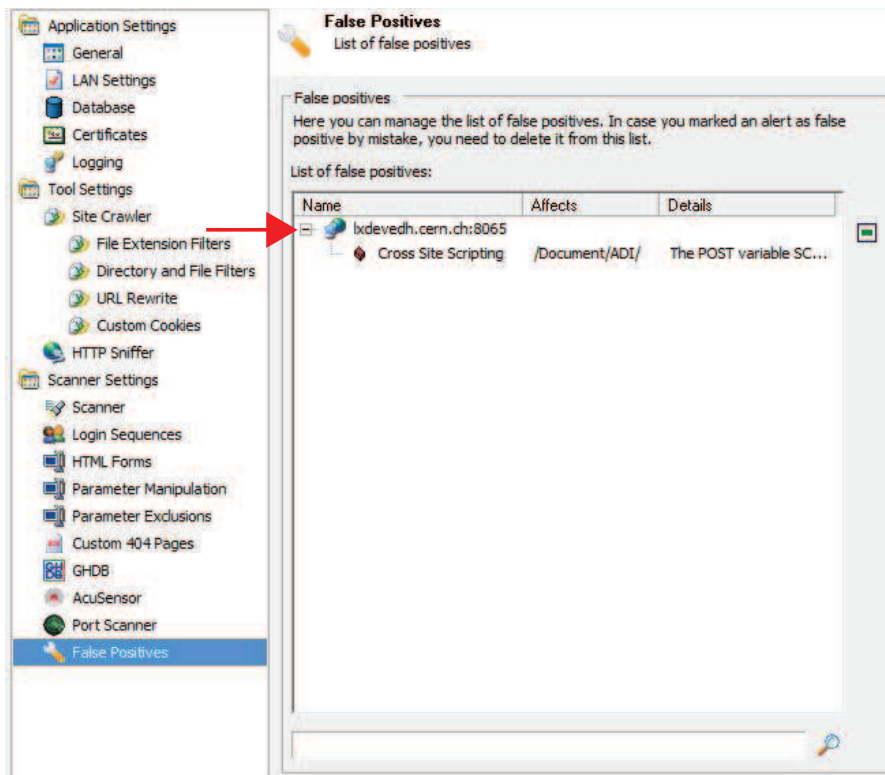


Figure 4.2: The false-positives are organized as a tree structure in Acunetix WWS (indicated by the red arrow). The figure shows the resulting false-positive tree after marking the SCROLLPOSN variable (in Figure 4.1) as false-positive.

Security Testing steps) is shown on the right hand side of the figure.

## 4.2 Integrating EAST in the AIS group's SDLC

Having selected a security testing methodology for Web applications based on certain defined criteria in Section 2.7, and described why the AIS group needs a new security testing methodology in their SDLC in Section 3.3, and further arrived at the desired security testing methodology (EAST) in Section 4.1, the author now integrates EAST in the SDLC applied by the AIS group. As mentioned in the problem statement (see Section 1.2), the integration is to be carried out at a proof of concept level. The following points explain where in Scrum (the SDLC applied by the AIS group) the steps of EAST are integrated.

**EAST step 1** (Misuse cases) is integrated in the Pregame phase, and can be executed during the creation of the product backlog items, and/or during the refining of a sprint backlog item into several backlog items.

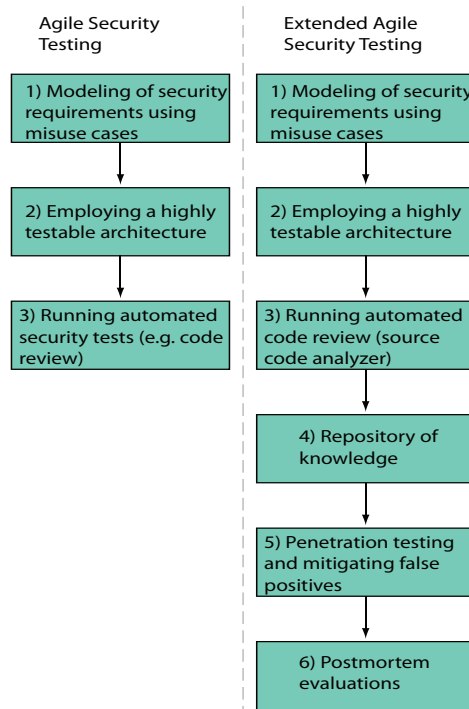


Figure 4.3: *The steps in Agile Security Testing (left hand side of the figure), and the steps in Extended Agile Security Testing (right hand side of the figure).*

**EAST step 2** (Employing a highly testable architecture) is integrated in the SDLC by integrating Misuse cases in the Pregame phase, Automatic code review in the Game phase, and Penetration testing and mitigating false-positives in the Game phase.

**EAST step 3** (Automatic code review) is integrated in the Game phase, and is to be executed in the Development phase.

**EAST step 4** (Repository of knowledge) is integrated in the Game phase and the Postgame phase. It is to be executed in the Development phase and the Review phase during the Game phase. Additionally, it is to be executed while the system is in operation as explained in Section 4.1.

**EAST step 5** (Penetration testing and mitigating false-positives) is integrated in the Game phase and is to be carried out in the Wrap phase after an executable version of the underlying backlog item is deployed.

**EAST step 6** (Postmortem evaluations) is integrated in the Game phase and is to be carried out during the Review phase, right after the penetration testing.

Figure 4.4 illustrates where in Scrum the EAST steps are to be executed. Furthermore, Section 5.1 explains *how*, *why* and by *whom* the EAST steps are to be carried out.

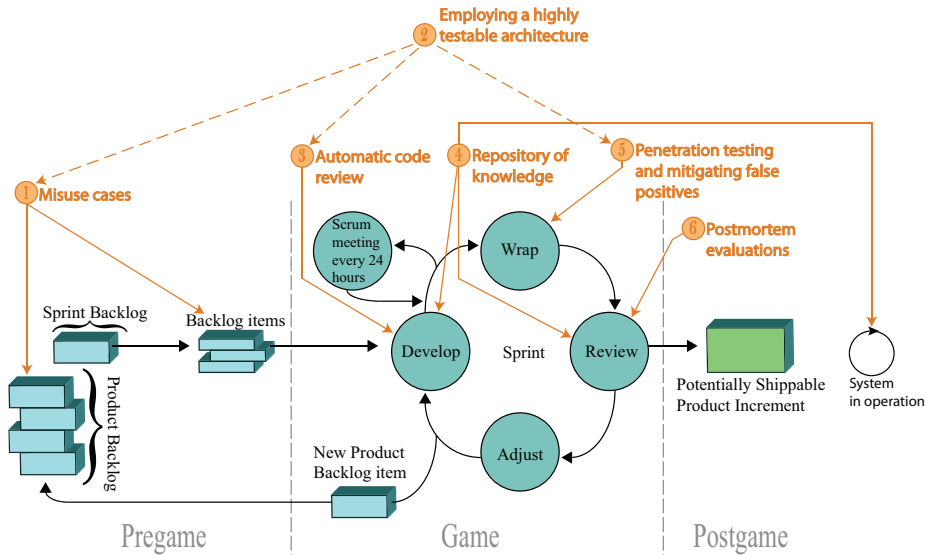


Figure 4.4: The Extended Agile Security Testing steps integrated in the appropriate phases of the SDLC applied by the AIS group.

### 4.3 Performing security tests

Security tests using the current security testing methodology (see Section 3.3) and using EAST needs to be carried out in order to measure the efficiency of EAST. By performing security tests using both methodologies and studying the results, and thereby looking at the difference, one can state whether EAST is more efficient than the current security testing methodology (see Figure 3.4).

However, conducting extensive tests using both methodologies is beyond the scope of this thesis due to time limitations. Therefore: (1) the security tests are limited to a certain amount of vulnerability classes, and (2) the security testing process is to be performed by using both security testing methodologies (current and EAST) two times (i.e. four security testing iterations).

Automated security tests can last for hours given the vast array of Web application vulnerabilities [12, 13, 7], and the vast array of automated vulnerability tests a Web Vulnerability Scanner can perform (see Table 2.3). The security tests are therefore limited to the OWASP Top 10 vulnerabilities, which are the following (as defined by OWASP) [36]:

- V.01 - Cross Site Scripting (XSS)** XSS flaws occur whenever an application takes user supplied data and sends it to a Web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface Web sites, possibly introduce worms, etc.
- V.02 - Injection Flaws** Injection flaws, particularly SQL injection, are common in Web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data.
- V.03 - Malicious File Execution** Code vulnerable to remote file inclusion (RFI) allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise. Malicious file execution attacks affect PHP, XML and any framework which accepts filenames or files from users.
- V.04 - Insecure Direct Object Reference** A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Attackers can manipulate those references to access other objects without authorization.
- V.05 - Cross Site Request Forgery (CSRF)** A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable Web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker. CSRF can be as powerful as the Web application that it attacks.
- V.06 - Information Leakage and Improper Error Handling** Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to steal sensitive data, or conduct more serious attacks.
- V.07 - Broken Authentication and Session Management** Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities.
- V.08 - Insecure Cryptographic Storage** Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers

use weakly protected data to conduct identity theft and other crimes, such as credit card fraud.

**V.09 - Insecure Communications** Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications.

**V.10 - Failure to Restrict URL Access** Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations by accessing those URLs directly.

It is not possible to test V.08 using automated security scanning tools [36]. As mentioned in the problem statement in Section 1.2, the security tests, using the the new security testing methodology (EAST), are to be carried out using a Web Vulnerability Scanner. V.08 is therefore excluded from the security tests in this thesis.

The data gathered by carrying out four security testing iterations is not enough to conclude whether EAST is more efficient than the current security testing methodology, but it gives a high level indication of the efficiency, which is sufficient for a proof of concept. As mentioned in Section 1.2, the testing scope is limited to parts of EDH. This is due to its size and complexity. The main functionalities that EDH provides are referred to as EDH documents. The EDH documents cover the following business areas:

- Administration
- Claims
- Human Resources & Training
- Leave
- Logistics
- Other Services
- Purchasing
- Safety

Figure 4.5 shows a screenshot of EDH and the abovementioned business areas covered by the EDH documents. There are several options (EDH documents) under each business area. E.g. under Other Services there is an option named Access Request. If an employee wants to get access to a restricted building, the employee must submit an Access Request for that specific building. After the submission, the people dealing with such requests will be notified through

EDH so they may either approve or deny the request. The security testing is to be carried out on two EDH documents. These are Internal Purchase Request (DAI) and Material Request (MAG), which are the most frequently used EDH documents. Figure 4.6 shows an activity diagram illustrating the execution of the security tests.



Figure 4.5: Screenshot of EDH showing the business areas covered by the EDH documents. The red arrows show where MAG and DAI are placed on the menu.

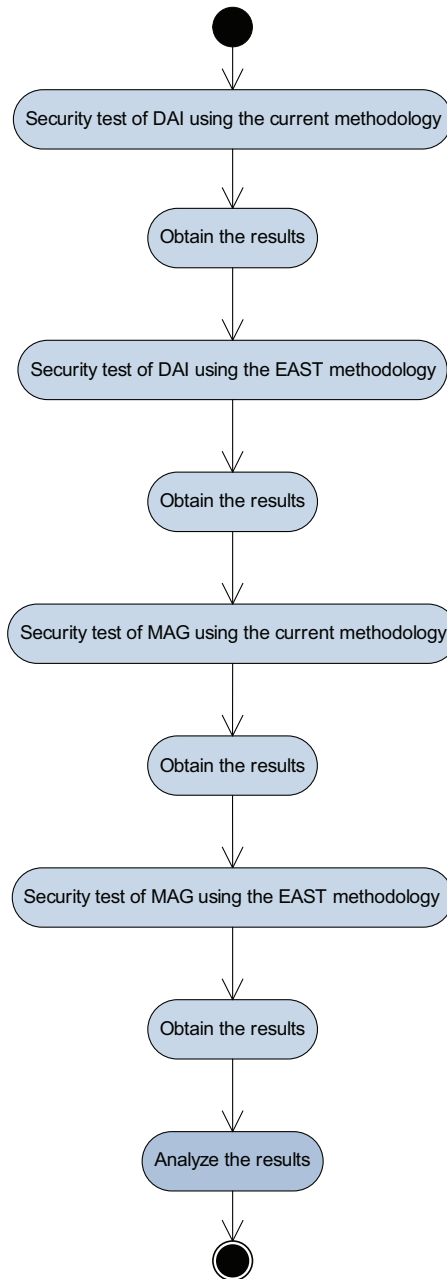


Figure 4.6: The testing processes are to be carried out two times using the current security testing methodology that is used at the AIS group, and two times using EAST. EDH document DAI is tested first by using the current methodology, then the results are obtained and collected, then DAI is tested again, but this time using EAST. The process is repeated for EDH document MAG. Finally, the results from both testing methodologies are analyzed and compared.



# Chapter 5

## Realization

This chapter describes how, why and by whom the EAST steps are to be carried out in the AIS group's SDLC as indicated in Figure 4.4. Further, it explains how the security tests on two EDH documents are carried out and describes the test results. The security tests are carried out using the current security testing methodology (see Figure 3.4), and using the EAST methodology (see Figure 4.4) on both of the EDH documents.

### 5.1 The integration of EAST: How, Why & Who

Section 4.2 described in which phases of Scrum the various EAST steps are integrated. This section describes how, why, and by whom the EAST steps are carried out in the specific Scrum phases as indicated in Figure 4.4. As mentioned in Section 2.7, there is a lack of empirical evaluations of security testing methodologies for Web applications. The author's explanation about who should carry out the EAST steps are therefore to some extent subjective.

- **EAST Step 1 (Misuse cases)**

- *How*: As explained in Section 3.2, the pregame phase in Scrum consists of planning and system architecture/high level design, which is carried out by creating product backlog items that are further refined into sprint backlog items. The creation of misuse cases are therefore executed in two steps in the pregame phase, in which step 1 is optional and step 2 is mandatory:

1. During the creation of product backlog items, high level misuse cases (i.e. misuse cases that contain high level specifications of the system) are created for each product backlog item. Since

the level of system specifications in a product backlog item is at a high level (not refined), the resulting misuse cases will also contain high level specifications. This step can be skipped if the product backlog item does not contain enough details in order to create misuse cases (e.g. missing details about the system architecture and design).

2. When a sprint backlog item is refined into several backlog items (indicated in the pregame phase in Figure 4.4), the system specifications are well defined and set up for development. The same transformation is to be applied on the misuse cases created in step 1 in order to create detailed misuse cases. If no misuse cases were created in step 1, they must be created containing detailed specifications of the system in this step. Finally, security requirements are to be derived using the resulting misuse cases.

- *Why*: Firstly, misuse cases let developers to think like an attacker (malicious user) and thereby enables them to get an overview of potential threats and vulnerabilities the evolving system may possess. Secondly, by using the misuse cases as a starting point the developers can create security requirements. The security requirements are then used to verify whether the system fulfills the required level of security (during the security testing process). Thirdly, the creative process of creating misuse cases let developers gain security specific knowledge. Last but not least, discovering vulnerabilities and creating countermeasures during definition, high level design, and low level design is the most cost efficient way of mitigating vulnerabilities (see Table 1.2).
- *Who*: Developers, system architects and security analysts should participate in developing misuse cases. Developers can look at the system from a low level viewpoint, while system architects can look at the system from a high level (design) viewpoint. Furthermore, since agile security testing doesn't completely eliminate the need for security experts (see Section 2.3.1), security experts are needed for misuse case completeness.

- **EAST Step 2 (Employing a highly testable architecture)**

- *How*: Unit security testing is achieved by performing automatic code review during the development phase (EAST step 3). System security testing is achieved by performing penetration testing after the creation of an executable version of the part or parts of the system

during the wrap phase (EAST step 5). Security acceptance testing is achieved by using the security requirements (EAST step 1) as reference points to verify or falsify the required level of security. Furthermore, EAST step 3 can be regarded as development testing, EAST step 5 can be regarded as system testing, and EAST step 1 can be regarded as basis for acceptance testing. This is also in line with the well known testing strategy; the V model (see Figure 2.9).

- *Why*: As explained in Section 2.3.1, a highly testable architecture is both useful for agile development methodologies and security testing. The highly testable architecture introduces test layers on top of the Web application layers as shown in Figure 2.8. It is therefore an architecture that suits agile development methodologies very well. Furthermore, it makes it possible to apply various security testing techniques within any number of the test layers (automatic code reviewing, penetration testing, etc.).
- *Who*: This step should be initiated by project managers. Project managers have an overview of the project development process, and can thereby initiate architects to employ a highly testable architecture.

- **EAST Step 3 (Automatic code review)**

- *How*: As described in Section 3.2, the development phase in Scrum consists of the following activities: Domain analysis, design, development, implementation, testing and documentation. Since testing is one of the activities, automatic code review is to be carried out in this phase while source code is being developed. I.e. for each unit (e.g. a class) the developer finishes, he or she must perform automatic code analysis on that particular unit. This is to be carried out using a static analysis tool.
- *Why*: By integrating automatic code review in the development phase, developers are able to correct the existing bugs at an early stage. Furthermore, this process continuously hardens the source code against security bugs. However, in order for this activity to be of maximum benefit, the developers need to have experience using the underlying static analysis tool and have security specific knowledge [59].
- *Who*: This step is carried out by developers.

- **EAST Step 4 (Repository of knowledge)**

– *How*: Every security decision that has been made during the development phase and the review phase must be documented. More specifically, this has to be done during the documentation activity in the development phase, and while reviewing potential risks in the review phase. The goal is to justify and document why certain security specific decisions were made. Additionally, the repository needs to be updated whenever a vulnerability is discovered while the system is in operation (illustrated by the arrow going from EAST step 4 to the “System in operation” loop in Figure 4.4). The level of detail on the justification may vary, but it must at least contain the following points:

- \* **Application**: The name of the application that the security decision applies for.
- \* **Decision ID**: An ID for the given security decision.
- \* **What**: A short explanation of what the security decision is.
- \* **Where**: The name of the affected part(s) of the application(s) due to the security decision (class(es), module(s), etc.).
- \* **How**: A short explanation of how the security decision is realized.
- \* **Why**: A short explanation of why the given security decision was made.

By providing answers to the abovementioned points it is possible to archive and store the security specific decisions that were made. Below is an example:

- \* **Application**: EDH
- \* **Decision ID**: EDH.SEC.001
- \* **What**: To ensure that sensitive information is not leaked whenever an error occurs.
- \* **Where**: EDH document A, B, and C.
- \* **How**: Show standard error message to the user by using the `StandardError()` class in the try-catch statements in the source code.
- \* **Why**: The `StandardError()` class contains predefined error messages that notifies the user that an error occurred without giving away detailed information, such as printing stack trace error message. E.g. “Error: the application has experienced a problem, please try later”. Giving away detailed error messages will be in

the benefit for a malicious user in order to obtain data about the system structure.

- *Why*: By documenting security specific decisions that has been made during development it is possible to keep the decisions in a repository, and thereby makes it possible to keep security specific knowledge alive within the organization. E.g. for training purposes, and for tracing earlier security specific decisions in order to understand why certain things are done the way they are. At first glance, this step may be regarded as a contradiction to one of the key thoughts in agile development, which is to document as little as possible. Agile software development values working software over comprehensive documentation [78], and tries to replace documents by initiating oral communication among developers, and initiating the usage of whiteboards etc. [62]. However, as mentioned in Section 2.3.1, there is a knowledge gap between security experts and software developers. Additionally, as mentioned in Section 4.1, there is a risk of losing years of knowledge when people quit their position. The author has therefore added this step in the EAST methodology in order to mitigate the knowledge gap and to mitigate the loss of security specific knowledge within the organization. Furthermore, looking at the abovementioned example, it is possible to see that such documentation of security specific decisions are not comprehensive, but rather a brief summary and justification of the underlying security decision.
- *Who*: Developers, system architects, project managers and other decision makers that have affected the security of the system must document why the security specific decisions have been made.

- **EAST Step 5 (Penetration testing and mitigating false-positives)**

- *How*: After an executable part or parts of a system is created in the wrap phase, a penetration test using a Web Vulnerability Scanner (see Table 2.3 for examples) has to be carried out on the executable part(s). The penetration testing results are then to be analyzed and the false-positives are to be marked. The false positives are to be marked as explained in Section 4.1.
- *Why*: By performing penetration tests in the wrap phase makes it possible to discover vulnerabilities in the application during a sprint (continuously). This creates a base for the review phase in which, among other things, risks are discussed, countermeasures are created and EAST Step 6 (Postmortem evaluations) is carried out. Fur-

thermore, by continuously marking false-positives, the application is harden against reporting many false-positives.

- *Who*: Developers, system architects and security analysts should participate in interpreting the penetration testing results produced by the penetration testing tool. By initiating security analysts (or security experts) in this step makes it easier to identify false-positives [59].

- **EAST Step 6 (Postmortem evaluations)**

- *How*: During the review phase there has to be a postmortem evaluation meeting session. The postmortem evaluation is to be carried out after the wrap phase and EAST Step 5 (Penetration testing and mitigating false-positives). Furthermore, it is to be carried out as explained in Section 4.1.
- *Why*: This step enables the security testers to reflect over the vulnerabilities, the development process and the security testing process. This is important in order to continuously harden the security testing process.
- *Who*: Everyone that have been involved in the security testing process should participate in a postmortem evaluation.

## 5.2 Security testing of EDH

As mentioned in Section 4.3, the security tests are to be performed on the most frequently used EDH documents: Internal Purchase Request (DAI) and Material Request (MAG). DAI is used to request materials from the internal stock at CERN, while MAG is used to register a purchase request in which external suppliers are contacted for the purchase. DAI and MAG communicate with different databases. The testing process is to be carried out as shown in the activity diagram in Figure 4.6. Figure 5.1 shows a screenshot of DAI and Figure 5.2 shows a screenshot of MAG.

Section 4.3 lists the vulnerability classes that are tested during the security testing of DAI and MAG, and for reasons given in the same section, the vulnerability class V.08 is excluded from the security testing process. Additionally, EDH is using Hypertext Transfer Protocol Secure (HTTPS) during *all* communication it has with a client. Therefore, V.09 Insecure Communications is also excluded from the testing process. Furthermore, some of the vulnerability classes are a collective term for different “types” of the same vulnerability class [102]:

- V.01 Cross Site Scripting (XSS) consists of:

- Reflected XSS
- Stored XSS
- DOM XSS
- Cross Site Flashing
- V.02 Injection Flaws consists of:
  - SQL Injection
  - LDAP Injection
  - ORM Injection
  - XML Injection
  - SSI Injection
  - XPath Injection
  - IMAP/SMTP Injection
  - Code Injection

It is beyond the scope of this thesis to test all the vulnerabilities in vulnerability class V.01 and vulnerability class V.02. The vulnerabilities that are to be tested are therefore limited to the nine vulnerabilities shown in Table 5.1. The following

Vulnerability Test	Description
VT.01	Reflected XSS
VT.02	Stored XSS
VT.03	SQL Injection
VT.04	Malicious File Execution
VT.05	Insecure Direct Object Reference
VT.06	Cross Site Request Forgery (CSRF)
VT.07	Information Leakage and Improper Error Handling
VT.08	Broken Authentication and Session Management
VT.09	Failure to Restrict URL Access

Table 5.1: *The vulnerabilities that are tested on DAI and MAG.*

points describe why reflected XSS, stored XSS and SQL injection were selected from vulnerability classes V.01 and V.02:

- Reflected XSS is tested because it is known to be the most frequent type of XSS attacks [80].
- Stored XSS is tested because it is known to be the most dangerous type of XSS attacks [102].
- SQL injection is tested because it is one of the most frequently applied attack type [56].

Furthermore, in order to get test results that are closely coupled to the testing methodologies (i.e. the current methodology and the EAST methodology), the tests are carried out according to the methodologies. However, the tests are also described and documented with traceability in mind. According to Koomen et al. [88], the first thing that needs to be created when testing software is a master test plan that sets up the total test process by:

- Aligning the test levels (the test levels given in the V model in Figure 2.9).
- Minimising overlaps or gaps in the test coverage.
- Optimal distribution of available resources, e.g.
  - Testers
  - Infrastructure and tools
  - Technical or domain expertise
- Detecting the most important defects at the earliest possible stage.
- Testing as early as possible on the critical path of the overall project.
- Achieving uniformity in the test process.
- Laying down agreements with stakeholders.
- Informing the client of the approach, planning, estimated effort, activities and deliverables in relation to the total test process.

Such a thorough test plan would lead to both traceability and would act as a roadmap to manage the whole testing process. However, creating such a thorough test plan for the purpose of this thesis would be an exaggeration. The author therefore systematically explains how the security tests are carried out in the following sections (which leads to traceability of the security tests):

- Section 5.2.1 describes the test environment, the testers' experience in security testing and their security specific knowledge.
- Section 5.2.2 describes how the security tests (given in Table 5.1), using the current methodology, are carried out by the testers. Furthermore, it describes the testing results each tester obtained (the amount of vulnerabilities found, and the amount of time spent for each vulnerability test).
- Section 5.2.3 describes how the security tests (given in Table 5.1) using the EAST methodology are carried out.



- Section 5.3 summarizes the test results obtained from the security tests performed on DAI and MAG.

Finally, it is necessary to mention that the time values for how much time it takes to complete one testing iteration, using either the current methodology or the EAST methodology, are gathered by observing the time that is spent on each testing activity in the methodology itself. Time values such as the time spent on introducing how the testers should use the guidelines given in Appendix A, and how to create misuse cases in Step 1 of the EAST methodology are not considered. Only the time spent on the activities per testing methodology are considered.

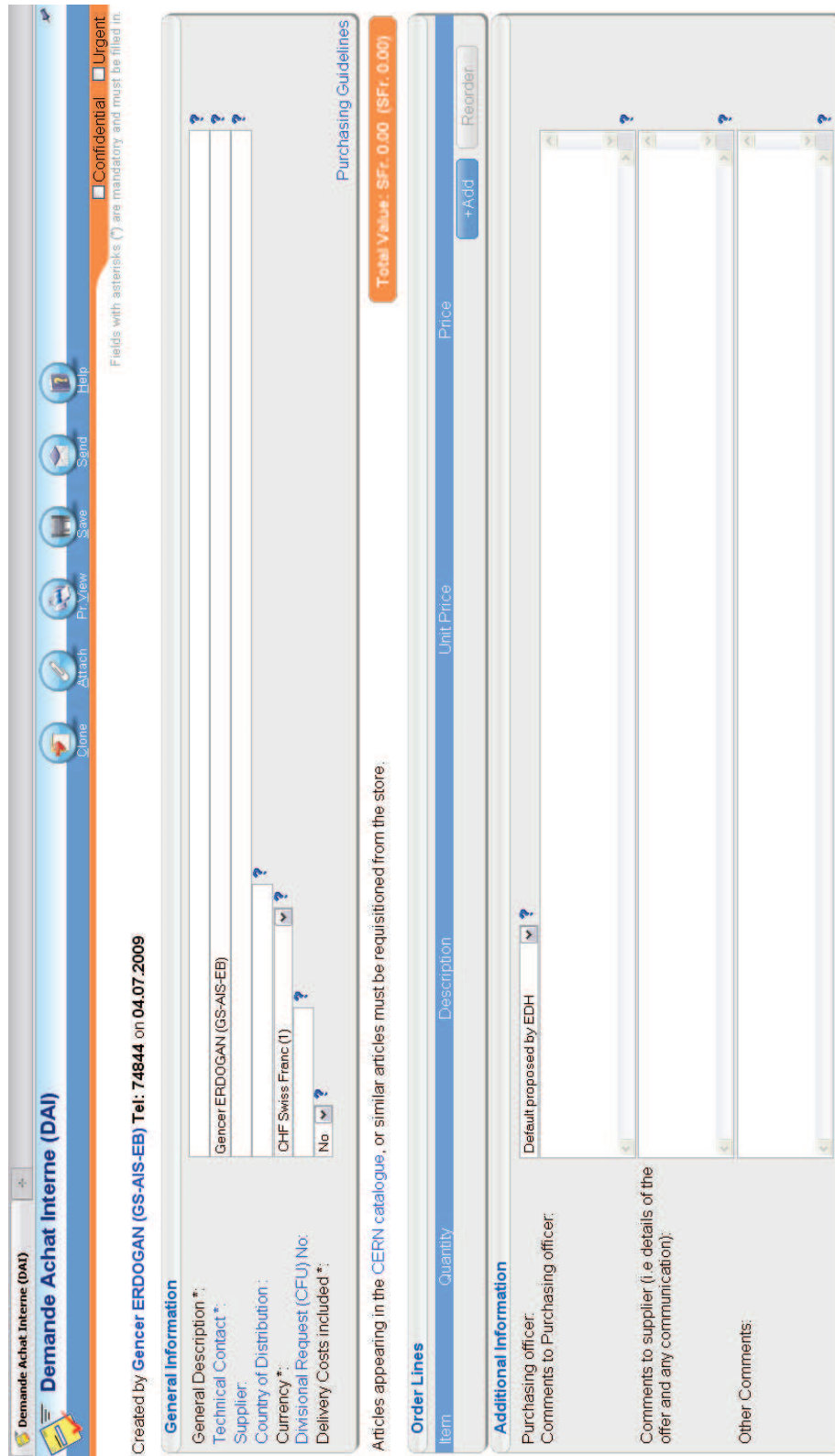


Figure 5.1: Screenshot of DAI as displayed to the user.

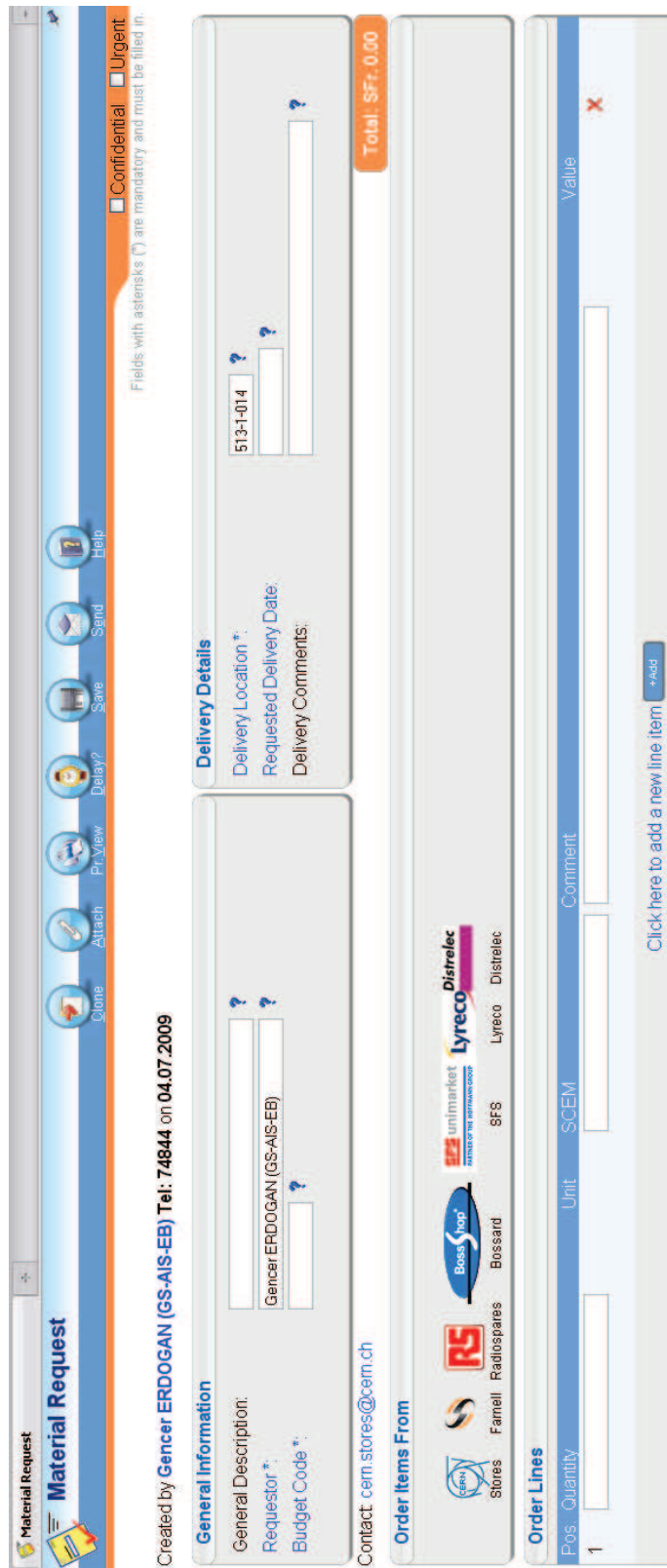


Figure 5.2: Screenshot of MAG as displayed to the user.

### 5.2.1 The test environment and the testers

A test environment is a composition of parts, such as hardware and software connections, environment data, maintenance tools and management processes in which a test is carried out [88].

Hardware refers to the physical parts of a computer (screen, harddisk, keyboard, network card, etc.). Software refers to the programs that should be present in order to perform the tests. Environment data is the set of data that is required by the software that is being tested (e.g. user profiles, data in the database the software communicates with and network addresses). Maintenance tools are the tools that are used to keep the test environment operational. Management processes are the activities that are carried out during the setup and maintenance of a test environment.

A test environment may vary in size. A small size test environment is e.g. when one single PC is used to test a small accounting package. A large size test environment is e.g. when a huge collection of hardware and software is used to test the reservation system of an airline company. The size of the test environment in this thesis is small. This is due to the small part of EDH that is tested (the EDH documents DAI and MAG). However, as mentioned in Section 5.2, DAI and MAG are the most frequently used EDH documents. Table 5.2 summarizes the test environment that is used to test for the vulnerabilities given in Table 5.1. The security tests given in Table 5.1 are carried out by three testers.

Environment part	Description
Hardware	Tester's working PC, EDH test server
Software	Internet Explorer or similar Web browser, Source code/text editor, PMD [39], SeaMonster [43]
Environment data	Test data available on EDH test server
Maintenance tools	None
Management processes	None

Table 5.2: *Summarization of the test environment. PMD is a static analyser tool that scans Java code for potential bugs. SeaMonster is a security modeling tool that supports, among other things, the modeling of misuse cases.*

All tests are performed on EDH's test server. In order to keep the anonymity of the testers, their names are not given and are referred to as Tester01, Tester02 and Tester03. Table 5.3 summarizes the testers' experience and knowledge.

### 5.2.2 Testing using the current methodology

This section provides answers to the following questions:

- How did each tester perform the specific vulnerability tests given in Ta-

Tester \ Characteristic	X years development experience	Security testing experience	Security specific knowledge
Tester01	8 years	Low	Medium
Tester02	3 years	Low	Low
Tester03	2 years	Low	Low

Table 5.3: Summarization of the testers’ experience and security specific knowledge. The possible values for the “Security testing experience” column and the “Security specific knowledge” column are: High, Medium or Low.

ble 5.1.

- How many vulnerability incidents for each specific vulnerability test were found by each tester?
- How many minutes did the tester spend on each vulnerability test?

The testers had access to the source code throughout the testing process, and each tester measured the time they spend on every security test. The functionalities of DAI and MAG are sometimes directly referred to in the points below (e.g. “the print preview button”). In order to understand where in the application GUI the functionalities are located, the reader is encouraged to look at Figure 5.1 for DAI, and Figure 5.2 for MAG.

The security test execution points below are given in the format “VTE.01”, “VTE.02” etc. This means Vulnerability Test Execution 01 (refers to VT.01 in Table 5.1) and Vulnerability Test Execution 02 (refers to VT.02 in Table 5.1) etc. Furthermore, the testers have executed the vulnerability tests in a quite similar manner. This is not surprising given that they all used the exactly same guidelines as starting point during the vulnerability tests (see Appendix A). Therefore, details regarding *how* the vulnerability tests were carried out are only given for Tester01. Anything that has been carried out differently than Tester01 is specified. Additionally, each of the testers performed the same testing strategy for MAG. I.e. Tester01 performed the same steps on MAG as performed on DAI (this is also true for Tester02 and Tester03). For this reason, only details about DAI is given. The results for MAG are listed in Section 5.3 together with the results for DAI.

### Testing using the current methodology on DAI

- **Tester01**
  - **VTE.01**: The following scripts were typed directly into the browser’s address line while inside EDH document DAI:

```
<script>alert(123);</script>
"><script>alert(123);</script><!--
```

which gives the following URLs:

```
https://lxdevedh.cern.ch:8065/Document/DAI/
<script>alert(123);</script>
```

```
https://lxdevedh.cern.ch:8065/Document/DAI/
"><script>alert(123);</script><!--
```

The scripts did not get executed from the URL. Source code for DAI was reviewed in order to find some variables that could be injected through the URL or input fields. 0 vulnerabilities found. 14 minutes spent.

- **VTE.02:** The following script was typed in the input fields (except the Currency field, the Delivery costs included field and the Purchasing officer field) of the DAI document:

```
<script>alert(document.cookie)</script>
```

The input fields interpreted the script as normal text when the Save button was pressed (the Save button submits the form). The source code was reviewed in order to see whether the input fields accepted the text from the input fields as they were typed or if they were converted into strings. The input fields were converted to strings (text). 0 vulnerabilities found. 10 minutes spent.

- **VTE.03:** Every input field is saved in a database. For every input field (except the Currency field, the Delivery costs included field and the Purchasing officer field) the following SQL query was entered (each input field were tested separately):

```
’; select count(1) from edhper;--
```

The “edhper” table name was found by reviewing the source code. 0 vulnerabilities found. 16 minutes spent.

- **VTE.04:** Created three Javascripts (test1.js, test2.txt and test3.SomeExtension). Each of the scripts contained the following code:

```
<script type="text/javascript">
alert(document.cookie);
</script>
```

The Attachment functionality was used to attach a file to the DAI document and the document was saved. By displaying the attachment list and clicking on each attachment, the Javascript described above was executed. Even if the Javascript file had different extensions, they got executed. 1 vulnerability found. 6 minutes spent.

- **VTE.05:** The DAI document does not have any identifiers (e.g. Document?DocumentNumber=0123456789) in the URL.
  1. The source code was reviewed to find if there are any variables that are set through the URL in the browser. None were found.
  2. The HTML code for the DAI document was downloaded to the local computer and reviewed to find hidden `<input>` variables. A hidden `<input>` variable (SCROLLPOSN) was found and modified from `type="hidden"` to `type="text"`.
  3. The `action` variable in the `<form>` tag was set to point to the DAI document.
  4. The form was submitted, but nothing notably happened. The DAI document appeared on the Web browser.

0 vulnerabilities found. 13 minutes spent.

- **VTE.06:** The HTML source code for the DAI document was reviewed in order to find some hidden variables that act as parameter identifier internally in the Web application. The hidden variable `TechnicalContact` was found. The tester created a simple Web site locally on his PC containing only the following link:

```
https://edh.cern.ch/Document/DAI/TechnicalContact=
Mister%20HACKER%20%28GS-HACKED-EB%29&
```

The tester logged in with his credentials on EDH, and then opened the simple Web site locally. By clicking the link on the Web site that the tester had created, he got redirected to a new DAI document, but the name of the Technical Contact field was not changed. The name was the tester's name and not "Mister HACKER (GS-HACKED-EB)" as indicated by the malicious link above. In this

test, the tester assumed both roles—the attacker and the victim. 0 vulnerabilities found. 18 minutes spent.

- **VTE.07:** For every input field in the DAI document, whenever a wrong input value was typed, the DAI document responded with a short error message for the issuing input field. None information leakage or improper error handling was detected from these error messages. Next, the tester deliberately made DAI crash by creating a null pointer in the source code for the DAI document and rebuilt the project. As expected, the DAI document crashed and as a result a stack trace of the null pointer exception was shown on the Web browser. This is a vulnerability. 1 vulnerability found. 15 minutes spent.
- **VTE.08:** The guidelines described for Broken Authentication and Session Management (VT.08) in Appendix A were followed and the following was done:
  1. Logged on EDH using Firefox Web browser.
  2. Used the AnEC [3] cookie management tool (add-on for Firefox) to obtain the session ID given by EDH.
  3. Logged out of EDH and deleted the cookie given by EDH.
  4. Restarted Firefox and logged on EDH.
  5. Logged out of EDH.
  6. Used the AnEC cookie management tool to replace the new session ID with the old session ID.
  7. Typed in `https://edh.cern.ch/Document/DAI/` in the browser.After step 7, the tester was able to navigate in EDH (using the old session ID) as if he was logged in successfully. However, for this to be a successful attack, the malicious user has to obtain a valid session ID from the victim. This can be done either by obtaining the session ID from the user's computer while the user is logged in EDH, or it requires some social engineering to convince the victim to give his/her session ID. This is therefore regarded as a vulnerability. 1 vulnerability found. 16 minutes spent.
- **VTE.09:** While in the following URL:

`https://edh.cern.ch/Document/DAI/`

the tester typed in names of both known and unknown directories after `Document/` and `Document/DAI/`. In the case of a correct directory and given that the tester have access to that specific directory,



the document was shown. Otherwise, an error message appeared saying either that the requested document did not exist, or that the tester did not have access to the given document. Examples of the directories:

```
https://edh.cern.ch/Document/DAI/ - Access granted.  
https://edh.cern.ch/Document/DAI/MEPPOverview/ - No access.  
https://edh.cern.ch/Document/DAI/something - Error, not found.  
https://edh.cern.ch/Document/something - Error, not found.  
https://edh.cern.ch/Document/../../../../ - Redirected to main  
page.
```

Additionally, the tester followed the guidelines and made a list of the links and where they pointed at in order to get a visual overview of the site structure. 0 vulnerabilities found. 12 minutes spent.

- **Tester02**

- **VTE.01:** Same approach as Tester01 (followed the guidelines), except reviewing source code. 0 vulnerabilities found. 5 minutes spent.
- **VTE.02:** Same approach as Tester01 (followed the guidelines), except reviewing source code. 0 vulnerabilities found. 8 minutes spent.
- **VTE.03:** Same approach as Tester01 (followed the guidelines), but instead of listing the query result, Tester02 tried to delete a row in a table using the following SQL query:

```
' ; delete from edhsupstate where supstateid='TEST'; --
```

The tester had inserted a test row in the target table. The query did not delete anything. 0 vulnerabilities found. 12 minutes spent.

- **VTE.04:** Same approach as Tester01 (followed the guidelines). 1 vulnerability found. 8 minutes spent.
- **VTE.05:** Same approach as Tester01 (followed the guidelines), except reviewing source code. The tester reviewed only the HTML code of the DAI document and followed the guidelines on how to change a hidden `<input>` variable. 0 vulnerabilities found. 10 minutes spent.
- **VTE.06:** Same approach as Tester01 (followed the guidelines). 0 vulnerabilities found. 14 minutes spent.
- **VTE.07:** Same approach as Tester01 (followed the guidelines), ex-

cept modifying source code (did not insert null pointer). 0 vulnerabilities found. 8 minutes spent.

- **VTE.08**: Same approach as Tester01 (followed the guidelines), but managed the cookie variables through Internet Explorer in stead of AnEC cookie management tool. 1 vulnerability found. 11 minutes spent.
- **VTE.09**: Same approach as Tester01 (followed the guidelines), 0 vulnerabilities found. 7 minutes spent.

- **Tester03**

- **VTE.01**: Same approach as Tester01 (followed the guidelines), except reviewing source code. 0 vulnerabilities found. 7 minutes spent.
- **VTE.02**: Same approach as Tester01 (followed the guidelines), except reviewing source code. 0 vulnerabilities found. 8 minutes spent.
- **VTE.03**: Same approach as Tester01 (followed the guidelines), but the tester tried to list several tables from the database and alter values in some tables. The tester used similar SQL query structure as Tester01 and Tester02. 0 vulnerabilities found. 14 minutes spent.
- **VTE.04**: Same approach as Tester01 (followed the guidelines). 1 vulnerability found. 9 minutes spent.
- **VTE.05**: Same approach as Tester01 (followed the guidelines). 0 vulnerabilities found. 11 minutes spent.
- **VTE.06**: Same approach as Tester01 (followed the guidelines). 0 vulnerabilities found. 13 minutes spent.
- **VTE.07**: Same approach as Tester01 (followed the guidelines), except modifying source code (did not insert null pointer). 0 vulnerabilities found. 10 minutes spent.
- **VTE.08**: Same approach as Tester01 (followed the guidelines), but managed the cookie variables through Internet Explorer in stead of AnEC cookie management tool. 1 vulnerability found. 13 minutes spent.
- **VTE.09**: Same approach as Tester01 (followed the guidelines). 0 vulnerabilities found. 9 minutes spent.

### 5.2.3 Testing using the EAST methodology

This section is divided in two parts: Testing DAI using the EAST methodology, and Testing MAG using the EAST methodology. For each part, the following points are given:

1. A misuse case diagram for the underlying EDH document (EAST Step 1).
2. The results from an automatic code review (using PMD via Eclipse) of the underlying EDH document (EAST Step 3).
3. A description of one security decision for the underlying EDH document. Any other security decisions are only listed describing what they cover. (EAST Step 4).
4. The results of the penetration testing and mitigating false-positives (using Acunetix WVS) for the underlying EDH document (EAST Step 5).
5. A list of answers to the following questions during the postmortem evaluation (EAST Step 6):
  - (a) Why were certain vulnerabilities missed during development?
  - (b) How will the issued development process be improved in order to mitigate or isolate the underlying vulnerabilities?
  - (c) Does a security testing activity (EAST Step 1 to EAST Step 6) need to be improved in order to detect the underlying vulnerabilities?

The details for how much time was spent on each of the abovementioned EAST steps are also given. Additionally, the details for how many vulnerabilities that were found from the automatic code review and the penetration testing are given. However, the measurement of how much time was spent for the abovementioned activities was not carried out in the same way as it was done for the current methodology (in Section 5.2.2). The EAST methodology is a collection of activities in which the testers have to collaborate. Furthermore, a collaboration sometimes require preliminary work which means that the testers have to do some work before the collaboration can take place. E.g. before the creation of misuse cases, each tester can come up with a list of threats and vulnerabilities. Each tester might therefore spend different amount of time depending on the testers experience and security specific knowledge. Then, the testers can present their findings and create misuse case diagrams jointly. This example has two different time measurements. The first measurement is how much time one tester spend on creating a list of threats and vulnerabilities. The second measurement is how much time the testers spend to create a misuse case diagram jointly. This is differently from the current methodology, in which a tester performs a set of vulnerability test without communicating with other testers (or sometimes do communicate with other testers). Apparently, this loose coupling of the communication between the testers is a side OEffect of ad hoc testing.

## Testing DAI using the EAST methodology

### 1. Misuse case diagram of DAI

The creation of the misuse case diagram for DAI was done in two steps:

1. Each tester made a list of possible threats and vulnerabilities that could occur in DAI.
2. The misuse case diagram was created jointly by the testers.

Figure 5.3 shows the resulting misuse case diagram. The following security requirements were derived from the misuse case diagram:

1. All of the input fields of a DAI document have to be checked for possible SQL injection whenever the document is saved. If any SQL injection is discovered, it has to be filtered out so that it is interpreted as normal text string for further processing.
2. All of the input fields of a DAI document have to be checked for possible scripting code whenever the document is saved. If any scripting code is discovered, it has to be filtered out so that it is interpreted as normal text string for further processing.
3. All of the previously saved input values in a DAI document have to be checked for possible scripting code whenever a saved DAI document is requested to be displayed by the user.
4. DAI must reject the document request if a user types in the full path to a DAI document in the browser's address bar in order to display the DAI document.
5. A specific DAI document can only be displayed on the user's browser if requested from the user's DAI document list.
6. Only the following document types can be allowed to be attached on a DAI document:
  - (a) gif
  - (b) jpg
  - (c) pdf
  - (d) doc
  - (e) xls
  - (f) docx
  - (g) xlsx

(h) txt

7. An attached document must be filtered, and only the file content is to be displayed, whenever it is requested to be displayed by a user.
8. A user's HTTP session ID must be invalidated whenever he logs out of a DAI document.
9. A user's HTTP session ID must be invalidated whenever he close the Web browser while displaying a DAI document.
10. A user's HTTP session ID must be renewed whenever he opens a new DAI document.
11. A DAI document must not leak internal code/file/folder information or stack trace information if an error occurs due to user input.
12. A DAI document must not leak internal code/file/folder information or stack trace information if an error occurs due to source code failure.

The following points list the time used by each tester to create a list of threats and vulnerabilities:

- Tester01 spent 20 minutes.
- Tester02 spent 15 minutes.
- Tester03 spent 18 minutes.

Additionally, the testers spent 23 minutes to create the misuse case diagram given in Figure 5.3 jointly.

## 2. Automatic code review of DAI

The static analyser tool, PMD, was used to perform automatic code review of the DAI source code. PMD is specially developed to review source code written in Java and is freely available at sourceforge [39]. Furthermore, it can be used as a standalone application, or it can be used as a plugin for, among other Integrated Development Environments (IDEs), Eclipse. The testers use Eclipse daily for software development and had previously used the PMD plugin for Eclipse. PMD was therefore chosen to be used for the automatic code reviewing. The automatic code review activity of DAI was done in two steps:

1. Each tester performed the automatic code review and analyzed the output generated by PMD.
2. The testers discussed whether there were any critical security issues from the findings.

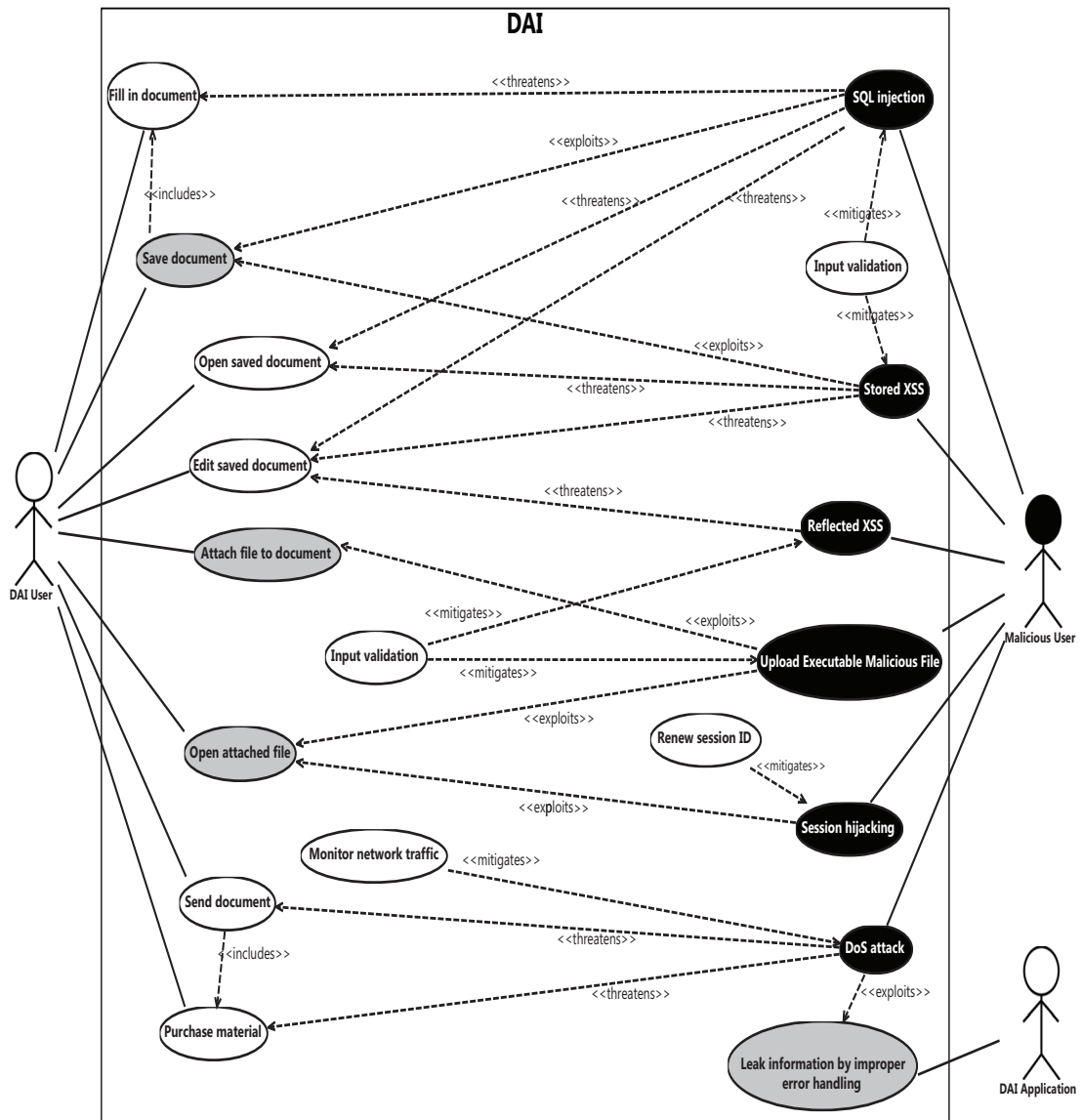


Figure 5.3: The resulting misuse case diagram for DAI. This misuse case diagram was created using SeaMonster [43].

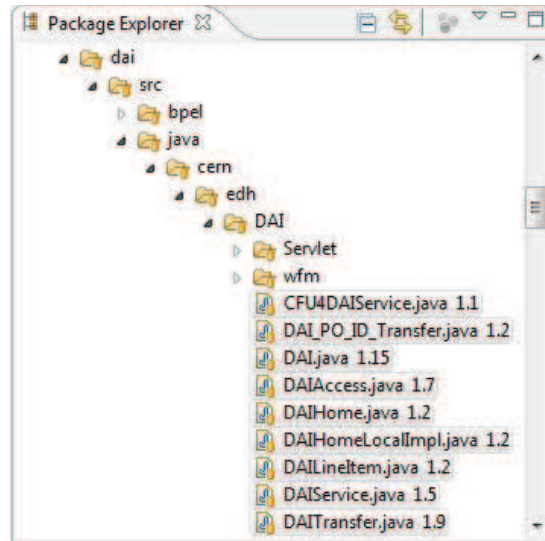


Figure 5.4: The folder structure of DAI and its nine Java source files that were scanned.

Figure 5.4 shows the folder structure of DAI and shows the Java files that were scanned. PMD has five levels of violations in which level one and two are regarded as errors, level three and four are regarded as warnings, and level five is regarded as informational warnings. Table 5.4 shows the total amount of errors, warnings and informational warnings PMD produced for the DAI source code. Figure 5.5 shows the PMD plugin for Eclipse in action. For each warning produced, the tester can click the specific warning to be directed to the issuing line in the source code. Furthermore, The tester can get detailed description of the warning by hovering the mouse over the warning sign. The following

Violation	Violations found
Error	36
Warning	426
Info	103

Table 5.4: The total amount of errors, warnings and informational warnings produced by PMD after scanning the DAI source code.

points list the time used by each tester to perform the automatic code review and to analyze the output generated by PMD:

- Tester01 spent 18 minutes.
- Tester02 spent 13 minutes.
- Tester03 spent 15 minutes.

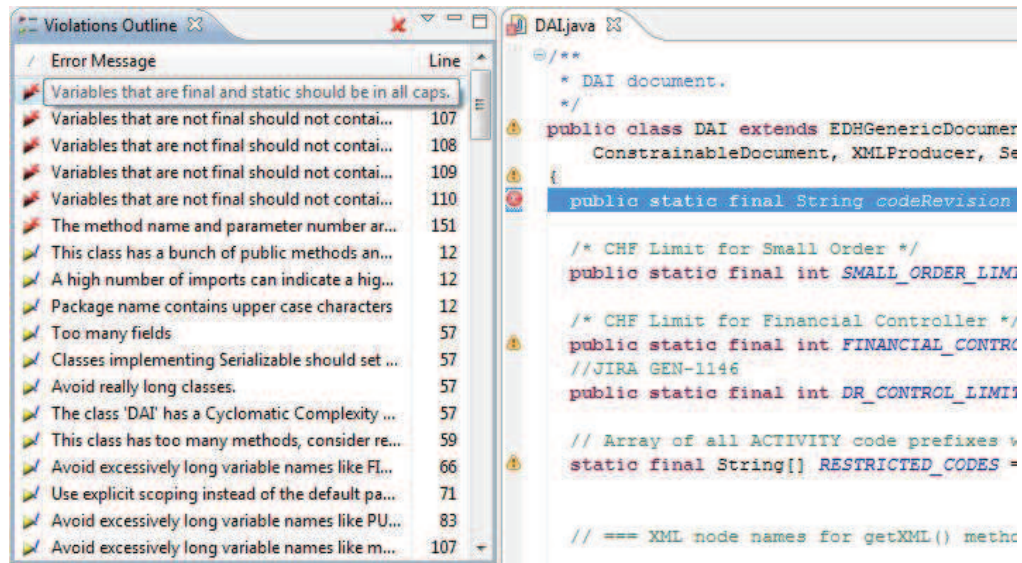


Figure 5.5: PMD lists all the warnings it generates and enables the tester to click on each specific warning in order to be navigated to the issuing line in the source code.

Additionally, the testers spent 20 minutes to discuss whether there were any critical security issues from the findings. Most of the warnings were regarded as guidelines for best programming practices by the testers. The testers concluded that there were no critical security issues. The two most critical warnings that PMD produced were the following: “Variables that are final and static should be in all caps.” and “Variables that are not final should not contain underscores (except for underscores in standard prefix/suffix).”

### 3. Security decisions (repository of knowledge) for DAI

Any security decisions that are made for the system that is being tested has to be documented during the development phase and the review phase, as explained in Section 5.1. Furthermore, a security decision has to be documented after the automatic code review (EAST Step 3), and after the penetration testing and mitigation of false-positives (EAST Step 5). The testers did not note any specific security decisions after the automatic code review, because there were no warnings from PMD that indicated a critical security related problem. The following security decisions were made after the penetration testing and mitigation of false-positives, and by using the misuse case diagram in Figure 5.3 as basis (only one security decision is explained in detail):

- *Application*: EDH.DAI
- *Decision ID*: EDH.DAI.001



- **What:** To mitigate SQL injections in DAI.
- **Where:** DAI source code.
- **How:**
  - Use only parametrized SQL queries.
  - Bind all dynamic data to parametrized SQL queries.
  - Never use string concatenation to create dynamic SQL queries.
- **Why:** SQL injection allows a malicious user to execute SQL queries within an application on the application's database. This is a very dangerous vulnerability and must be mitigated.

Furthermore, the following security decisions were made:

- To mitigate Reflected Cross Site Scripting in DAI.
- To mitigate Stored Cross Site Scripting in DAI.
- To mitigate the execution of attached malicious files in DAI.
- To mitigate session hijacking in DAI.
- To mitigate DoS attacks on EDH. This security decision covers EDH as total and not only DAI or MAG.

The creation of security decisions for DAI was carried out jointly by the testers. The total amount of time spent was 28 minutes.

#### 4. Penetration testing and mitigating false-positives of DAI

The penetration testing of DAI was carried out using Acunetix WVS. As mentioned in Section 2.5, Acunetix WVS allows users to create customized scan profiles in order to perform specific security tests and thereby reduce the total scan time. Before the penetration testing took place, a scan profile for each of the vulnerabilities mentioned in Table 5.1 was created. Figure 5.6 shows the profile for Reflected XSS created in Acunetix WVS. The penetration testing results are listed in Table 5.5. The penetration testing results show that 1 Reflected XSS, 43 SQL injections and 10 possible Cross Site Request Forgeries (CSRF) were detected by Acunetix WVS. The 10 CSRFs were not reported as vulnerabilities, but as informational warnings and are therefore not added to the total vulnerabilities found in Table 5.5. The testers analyzed the output and concluded that the 10 informational warnings were false-positives and therefore marked as false-positives in Acunetix WVS. Figure 5.7 shows the resulting false-positive tree generated by Acunetix WVS. The penetration testing lasted

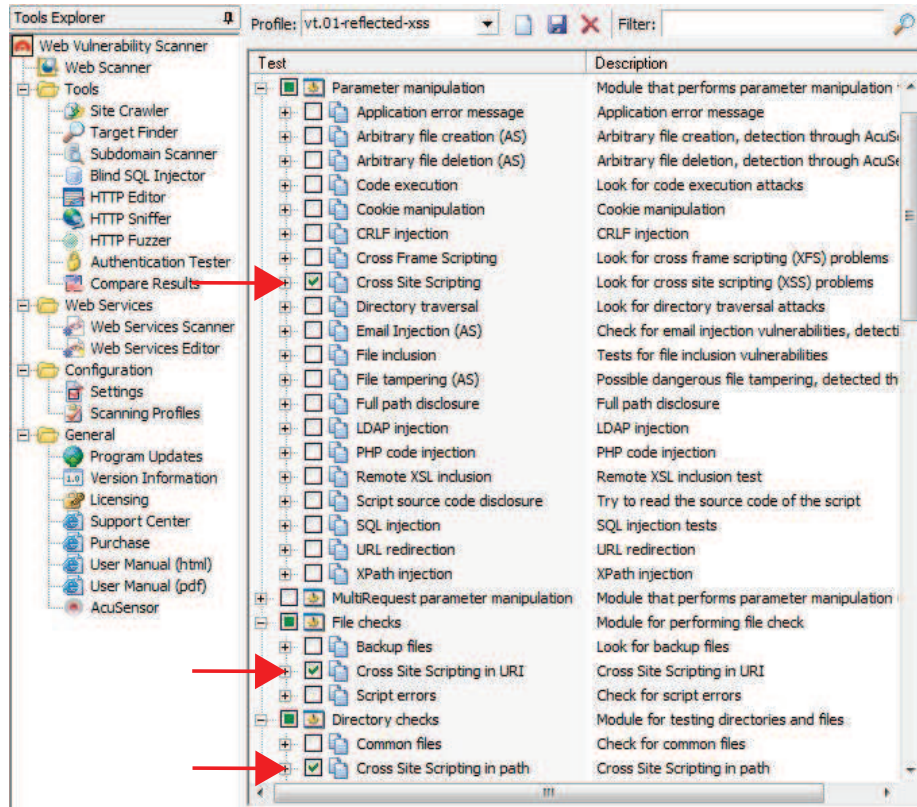


Figure 5.6: The Reflected XSS profile created in Acunetix WVS. When scanning a Web application using this profile, Acunetix WVS will scan for the tests indicated by the red arrows.

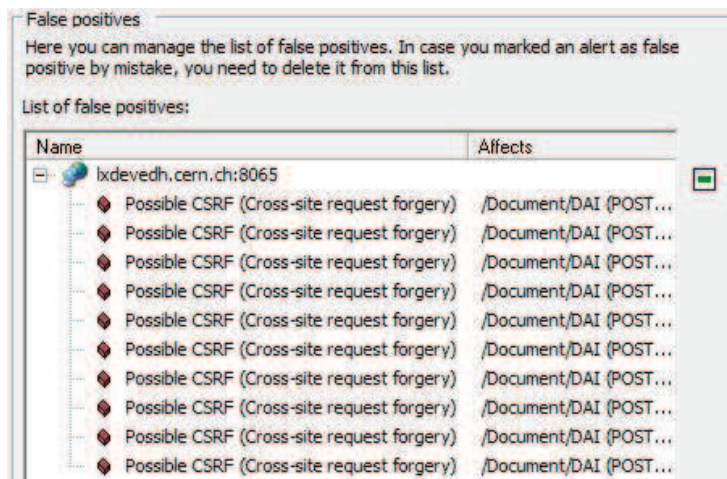


Figure 5.7: The resulting false positive tree after marking all of the 10 possible CSRFs as false-positives in Acunetix WVS.

Vulnerability	Vulnerabilities found	Minutes spent	Seconds spent
VT.01	1	1	30
VT.02		1	5
VT.03	43	4	46
VT.04		1	11
VT.05		5	45
VT.06	10 info.		11
VT.07		5	22
VT.08			36
VT.09			28
<b>Total</b>	44	17	234
The total time spent is: 20.90min $\approx$ 21min.			

Table 5.5: The penetration testing results of DAI. 1 Reflected XSS, 43 SQL injections and 10 possible (informational) Cross Site Request Forgeries were reported.

21 minutes, and the testers spent 25 minutes analyzing the penetration testing output. This gives a total time of 46 minutes.

## 5. Postmortem evaluations for DAI

The postmortem evaluation took place right after the penetration testing and the creation of security decisions. The testers discussed and provided (to the best of their ability) answers to the questions defined at the start of this section. These questions are the same as those that are defined in Section 4.1 (point 2). The following points shortly give the answers provided by the testers:

- *Answer to question 1:* The fact that no security testing practices are considered before and during development in the current security testing methodology, is an obvious indication to why the vulnerabilities listed in Table 5.5 were missed during development. Furthermore, using the EAST methodology to test DAI, lead to the detection of the vulnerabilities that had been introduced in DAI as a result of the current security testing methodology. However, it is possible to see from the misuse case diagram in Figure 5.3 that countermeasures for the vulnerabilities that were found for DAI has been considered. The vulnerabilities that were found would therefore be mitigated during the development process, if DAI was to be developed using the EAST methodology as a starting point.
- *Answer to question 2:* Creating misuse case diagrams in the pregame phase (in Scrum) would definitely alert the developers about possible threats and vulnerabilities (e.g. Cross Site Scripting and SQL injections) that may arise in the evolving system. Furthermore, by using the docu-

mentation for the security decisions (produced by EAST Step 4 - Repository of knowledge) in the pregame and game phases, knowledge management regarding security knowledge can be obtained.

- *Answer to question 3:* There is currently too little experience in using the EAST methodology and its steps in order to say whether some of the steps need to be improved or not.

The postmortem evaluation of DAI was carried out jointly by the testers. The total amount of time spent was 35 minutes.

### Testing MAG using the EAST methodology

#### 1. Misuse case of MAG

The creation of the misuse case diagram for MAG was done in the same way as for DAI. The resulting misuse case for MAG is the same as for DAI. This is due to the similarity between the functionalities provided by DAI and MAG. However, MAG has one functionality that differs from DAI, which is to provide links to external supplier (see the Order Items From section in Figure 5.2). Whenever a link to an external supplier is clicked, MAG automatically redirects the user to the supplier's Web site. This is a function that needs to be considered from a security viewpoint. If the security quality of the supplier's Web site has been compromised, then a MAG user might be a victim. But the security quality of an external supplier's Web site is beyond the "border" of MAG's (and consequently EHD's) concern. The following points list the time used by each tester to create a list of threats and vulnerabilities:

- Tester01 spent 12 minutes.
- Tester02 spent 10 minutes.
- Tester03 spent 10 minutes.

Additionally, the testers spent 15 minutes to create the misuse case diagram jointly, but it was quickly noticed that the misuse case diagram for MAG was the same as for DAI.

#### 2. Automatic code review of MAG

Automatic code reviewing of MAG was carried out in the same way as explained for DAI. Table 5.6 shows the total amount of errors, warnings and informational warnings PMD produced for the MAG source code. The following points list the time used by each tester to perform the automatic code review and to analyze the output generated by PMD:

Violation	Violations found
Error	56
Warning	498
Info	106

Table 5.6: *The total amount of errors, warnings and informational warnings produced by PMD after scanning the MAG source code.*

- Tester01 spent 14 minutes.
- Tester02 spent 10 minutes.
- Tester03 spent 13 minutes.

Additionally, the testers spent 22 minutes to discuss whether there were any critical security issues from the findings.

### 3. Security decisions (repository of knowledge) for MAG

The creation of security decisions for MAG was carried out in the same way as explained for DAI. Not surprisingly, the decisions that the testers ended up with were the same as for DAI because of the similarities between DAI and MAG. Instead of defining new security decisions, the testers referred to those created for DAI by using their Decision ID. E.g. *Decision ID: EDH.DAI.001*. The creation of security decisions for MAG was carried out jointly by the testers. The total amount of time spent was 15 minutes.

### 4. Penetration testing and mitigating false-positives of MAG

The penetration testing for MAG was carried out in the same way as explained for DAI. The penetration testing results for MAG are listed in Table 5.7. The 2 CSRFs were not reported as vulnerabilities, but as informational warnings and are therefore not added to the total vulnerabilities found in Table 5.7. As done for ADI, the CSRFs were analysed and marked as false-positives. The penetration testing lasted 8 minutes, and the testers spent 16 minutes analyzing the penetration testing output. This gives a total time of 24 minutes.

### 5. Postmortem evaluations for MAG

The postmortem evaluations for MAG was carried out in the same way as explained for DAI. Furthermore, the resulting answers that the testers provided to the questions (the questions defined at the start of this section) ended up being the same as the answers given for DAI. This is again due to the similarities between DAI and MAG. Although the result ended up being the same as given for DAI, the postmortem evaluation session was carried out by answering each question. The postmortem evaluation of MAG was carried out jointly by the

Vulnerability	Vulnerabilities found	Minutes spent	Seconds spent
VT.01	10		47
VT.02			7
VT.03	3	1	22
VT.04			24
VT.05		2	15
VT.06	2 info.		9
VT.07		2	2
VT.08			22
VT.09			36
<b>Total</b>	13	5	184
The total time spent is: $8.06min \approx 8min$ .			

Table 5.7: The penetration testing results of MAG. 10 Reflected XSS, 3 SQL injections and 2 possible (informational) Cross Site Request Forgeries were reported.

testers. The total amount of time spent was 15 minutes.

## 5.3 Test results

The following sections summarize the results obtained from testing DAI and MAG using both the current methodology and the EAST methodology. Section 5.3.1 summarizes the results obtained using the current methodology, while Section 5.3.2 summarizes the results obtained using the EAST methodology.

### 5.3.1 The current methodology test results

Table 5.8 shows the results obtained for DAI, while Table 5.9 shows the results obtained for MAG. Both of the tables reveal that the testers found one VT.04 vulnerability (Malicious File Execution) and one VT.08 vulnerability (Broken Authentication and Session Management). Additionally, Tester01 found one VT.07 vulnerability (Information Leakage and Improper Error Handling). In total, 3 different vulnerabilities were found by the testers using the current testing methodology. I.e. for VT.04 and VT.08, the testers found the same vulnerabilities, not different incidents of the same vulnerability class.

### 5.3.2 The EAST methodology test results

Table 5.10 shows the measured time results obtained by testing DAI using the EAST methodology, while Table 5.11 shows the measured time results obtained

<b>DAI</b>						
	Tester01		Tester02		Tester03	
VTE	TS	VF	TS	VF	TS	VF
VTE.01	14 min		5 min		7 min	
VTE.02	10 min		8 min		8 min	
VTE.03	16 min		12 min		14 min	
VTE.04	6 min	1	8 min	1	9 min	1
VTE.05	13 min		10 min		11 min	
VTE.06	18 min		14 min		13 min	
VTE.07	15 min	1	8 min		10 min	
VTE.08	16 min	1	11 min	1	13 min	1
VTE.09	12 min		7 min		9 min	
Total	120 min	3	83 min	2	94 min	2
The average time spent is: $297/3 = 99min$ .						

Table 5.8: The test results obtained by testing DAI using the current methodology. VTE = Vulnerability Test Execution, TS = Time Spent, VF = Vulnerabilities Found.

<b>MAG</b>						
	Tester01		Tester02		Tester03	
VTE	TS	VF	TS	VF	TS	VF
VTE.01	6 min		4 min		5 min	
VTE.02	9 min		6 min		5 min	
VTE.03	12 min		10 min		8 min	
VTE.04	4 min	1	5 min	1	6 min	1
VTE.05	11 min		8 min		10 min	
VTE.06	10 min		9 min		10 min	
VTE.07	12 min	1	5 min		8 min	
VTE.08	13 min	1	10 min	1	12 min	1
VTE.09	10 min		6 min		8 min	
Total	87 min	3	63 min	2	72 min	2
The average time spent is: $222/3 = 74min$ .						

Table 5.9: The test results obtained by testing MAG using the current methodology. VTE = Vulnerability Test Execution, TS = Time Spent, VF = Vulnerabilities Found.

by testing MAG using the EAST methodology. As shown in Table 5.5 and Table 5.7, the total amount of reported vulnerabilities are 54 for DAI and 15 for MAG respectively. 12 of the reported vulnerabilities were marked as false-positives. Furthermore, the one VT.01 vulnerability (Reflected XSS) found for DAI was the same as one of the VT.01 vulnerabilities found for MAG. This gives a total of 10 different VT.01 vulnerabilities and a total of 46 different VT.03 (SQL Injection) vulnerabilities (see Table 5.12).

## DAI

EAST	Tester	Tester01	Tester02	Tester03	Jointly
	Step 1		20 min	15 min	18 min
Step 3		18 min	13 min	15 min	20 min
Step 4					28 min
Step 5					46 min
Step 6					35 min
Total		38 min	28 min	33 min	152 min
The average time spent is: $251/3 = 83.67min.$					

Table 5.10: The measured time results obtained by testing DAI using the EAST methodology.

## MAG

EAST	Tester	Tester01	Tester02	Tester03	Jointly
	Step 1		12 min	10 min	10 min
Step 3		14 min	10 min	13 min	22 min
Step 4					15 min
Step 5					24 min
Step 6					15 min
Total		26 min	20 min	23 min	91 min
The average time spent is: $160/3 = 53.33min.$					

Table 5.11: The measured time results obtained by testing MAG using the EAST methodology.

Vulnerability	EDH Document	DAI	MAG	In com- mon	Total
	VT.01		1	10	1
VT.03		43	3	0	46
The total number of different vulnerabilities found is 56.					

Table 5.12: The total vulnerabilities found for DAI and MAG by testing DAI and MAG using the EAST methodology



# Chapter 6

## Evaluation and discussion

This chapter uses the test results described in Section 5.3 as basis to compare the efficiency of the current security testing methodology and the EAST methodology. Then it describes some threats to the validity of the test results. Furthermore, it gives an explanation of how the goals defined in the problem statement in Section 1.2 are fulfilled, and whether the research goals defined in Section 1.3 are attained. Finally, it gives an explanation on whether the hypothesis defined in Section 1.4 is verified or falsified.

### 6.1 Comparison of the methodologies

There are three factors that are used as basis to compare the efficiency of the current security testing methodology and the EAST methodology. These factors are also mentioned in the problem statement and in the initial hypothesis in Chapter 1, which are:

1. The amount of time spent on the security testing process.
2. The amount of vulnerabilities found during the security testing process.
3. The ability to mitigate false-positives during the security testing process.

Figure 6.1 shows an overview the test results obtained from each test iterate (as shown in the activity diagram in Figure 4.6). Studying Table 5.8, 5.9, 5.10, 5.11 and 5.12 reveals the following facts:

#### 1. Efficiency regarding the amount of time spent:

- The average time spent on testing DAI using the current methodology is *99 minutes*, while the average time spent on testing DAI using the EAST

methodology is 83.67 *minutes*. This shows that the EAST methodology is approximately **15%** more effective than the current methodology regarding the average time spent on testing DAI only.

- The average time spent on testing MAG using the current methodology is 74 *minutes*, while the average time spent on testing MAG using the EAST methodology is 53.33 *minutes*. This shows that the EAST methodology is approximately **28%** more effective than the current methodology regarding the average time spent on testing MAG only.
- By comparing the average time spent on testing both DAI and MAG using the current methodology versus the average time spent on testing both DAI and MAG using the EAST methodology, the following equation is obtained:

$$\frac{100\%}{99.00min + 74.00min} (83.67min + 53.33min) \approx 79\%$$

which means that the average time spent on testing both DAI and MAG using the EAST methodology is 79% of the average time spent on testing both DAI and MAG using the current methodology. Hence, the EAST methodology is approximately **21%** more effective in average, regarding time spent, than the current methodology (see Equation (6.1)).

$$100\% - 79\% = \mathbf{21\%} \quad (6.1)$$

## 2. Efficiency regarding the amount of vulnerabilities found:

The amount of vulnerabilities found by testing DAI and MAG using the current methodology is 3, while the amount of vulnerabilities found by testing DAI and MAG using the EAST methodology is 56. In this respect, the EAST methodology is approximately **95%** more effective than the current methodology (see Equation (6.2)).

$$100\% - \frac{100\%}{56vulnerabilities} (3vulnerabilities) \approx \mathbf{95\%} \quad (6.2)$$

However, the three vulnerabilities that were found by using the current methodology were not found by using the EAST methodology. This indicates that human intervention is necessary in circumstances where tools have limited capabilities. Furthermore, this is in line with the statements given about the shortcomings of agile security testing in Section 2.3.1. I.e., agile security testing does not eliminate the need for security experts and has test incompleteness.

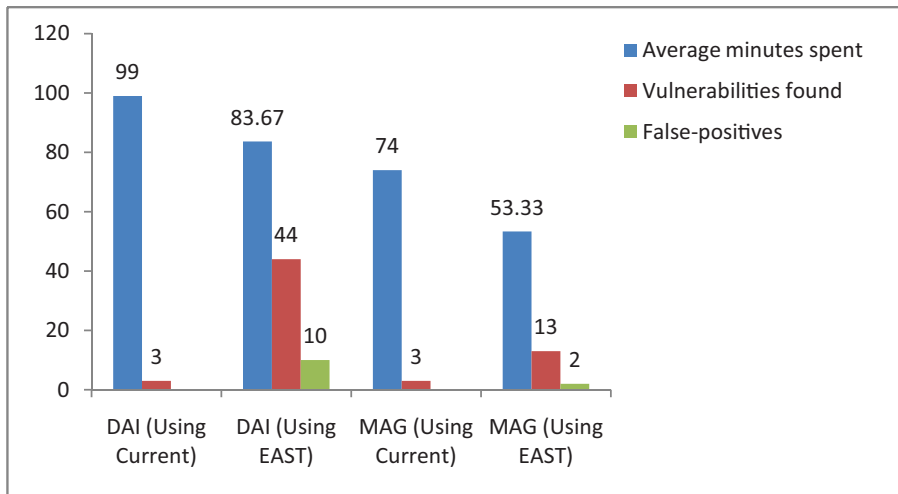


Figure 6.1: Comparison of the test results.

### 3. The ability to mitigate false-positives:

By testing DAI and MAG using the EAST methodology (and Acunetix WVS), there were found 12 false-positives. The false-positives were marked via Acunetix WVS and thereby added to a false-positive repository. Next time DAI and MAG are scanned, the marked false-positives will be regarded as false-positives and not as vulnerabilities by Acunetix WVS. In turn, this mitigates false-positives during the security testing process. The current security testing methodology does not have an activity in place that mitigates false-positives. Hence, only the EAST methodology mitigates false-positives during the security testing process.

However, as mentioned in Section 4.1, this approach to false-positive mitigation is dependent on either: (1) the penetration testing tool (that is being used in the EAST methodology) must have the ability to mark and remember specific false-positives or, (2) the penetration testing tool must have the ability to import a false-positive repository (e.g. false-positive database, XML file, etc.).

## 6.2 Threats to validity

The following points describe the threats to the generalizability of the test results:

- **Tools:** The usage of one specific **Web Vulnerability Scanner** is obviously an important factor that affects the results of the tests performed in Section 5.2.3. Other Web Vulnerability Scanners could produce different results regarding time spent and vulnerabilities found. However, this risk is mitigated due to the reasons given in Section 2.5.1 (Why Acunetix

WVS?). Furthermore, it was decided that **PMD** would be used as the automatic code scanner tool. This was due to the testers' previous experience in using PMD. A better starting point in this respect would be to find a tool that produce the minimum amount of false-positives. This risk is mitigated by leaning towards the findings done by Baca et al. [59]. I.e. the best security detectors are those that have both security experience and experience in using the static analyzer tool being used. Finally, the testers used **SeaMonster** to model misuse case diagrams. Although the testers had not used SeaMonster previously, they did get an introduction on how to use it. However, the lack of experience in using SeaMonster would inevitably affect the test results regarding time spent.

- **Security specific knowledge:** The majority of the testers had low security specific knowledge (see Table 5.3). This would consequently lead to the detection of less vulnerabilities (as shown in Table 5.8 and Table 5.9).
- **Security testing experience:** All of the testers had a low experience in security testing. This can also be used as a basis to question the validity of the test results. On the other hand, this is another example showing that structured security testing is still not widely applied in organizations. This risk can only be mitigated by continuously applying structured security testing using e.g. the EAST methodology.
- **Testing thoroughness:** The thoroughness of the testers' test execution was not measured, except the given explanation on how the testers did perform the test activities in each security testing methodology (Section 5.2.2 and Section 5.2.3). This risk is mitigated (or believed to be mitigated) by the fact that there were only voluntarily testers.
- **Psychological effects, and the number of testers:** As shown in Figure 4.6, the test iterations were carried out sequentially. This would have a learning effect on the testers. I.e. for each iterate, the testers would gain more security specific knowledge and more experience in performing the tests. Obviously, this would be in the benefit of the testers, which would further produce better results. In this context, better results means spending less time to complete a test iterate and/or to find more vulnerabilities. Although the testers knew that the purpose of the testing process was to observe the testing methodologies, they may have been more effective while performing the security tests and thereby improving the results, as a response to the fact that they were the ones that produced the results that were taken into consideration. Thus, the testers may have considered themselves as a part of the "object" being observed and thereby improved

the test results. This form of reactivity is referred to as the Hawthorne effect. Figure 6.1 shows that the average time spent on the test iterations decreases for each test iterate. This indicates that there has indeed been a learning effect on the testers during the testing process. Furthermore, the number of testers is another factor that could be used to question the validity of the test results. However, this was a side effect of the limited time and resources.

## 6.3 Goals attained

This section starts by explaining how the goals defined in the problem statement in Section 1.2 are attained, i.e. how the problem statement is solved. It further explains whether the research goals defined in Section 1.3 are attained. Section 6.3.1 explains the former, while Section 6.3.2 explains the latter. The problem statement points and the research goal points are referred to by their identifiers, (e.g. Problem statement point 1, and RG.01). The reader is therefore encouraged to look at Section 1.2 for the problem statement, and Section 1.3 for the research goals in order to understand the points that are being referred to.

### 6.3.1 Problem statement goals

- **Problem statement point 1:** In Chapter 2, the author performed a survey of state-of-the-art security testing methodologies, with a special focus on Web applications. This was done by gathering and analyzing relevant information from academic books, scientific papers and Web sites. With this, problem statement point 1 is fulfilled.
- **Problem statement point 2:** The author defined a set of criterions in Section 2.7. Using those criterions as basis, the author elicited Agile Security Testing as the most adequate security testing methodology for Web applications. However, for reasons given in Section 2.7, the author's selection criteria are to some extent subjective. In turn, this could have an affect on the resulting decision.

Initially, Agile Security Testing did only fulfill point 2a in the problem statement, and did not fulfill point 2b. Neither did it fulfill criterion C.05 and criterion C.06 defined in Section 2.7. Therefore, the author provided a solution for point 2b, criterion C.05 and criterion C.06 as shown in Section 4.1. The resulting solution was named Extended Agile Security Testing (EAST). With this, problem statement point 2 is fulfilled.

- **Problem statement point 3:** In Chapter 3, the author presented the current situation of the AIS group at CERN, and described the SDLC that is being used by the AIS group (Scrum). Furthermore, an explanation to why there is a need for a security testing methodology in the AIS group was given. This was done by:
  1. Describing the security testing methodology applied by the AIS group, and giving an explanation to why the current security testing methodology is not sufficient.
  2. Conducting a risk analysis of the AIS group's software systems, by using the most important assets that are handled by their software systems as a starting point.

In Section 4.2 the author described in which phase of Scrum the various EAST steps are integrated. Finally, in Section 5.1 the author completed the integration by describing how, why, and by whom the EAST steps are carried out. With this, problem statement point 3 is fulfilled.

- **Problem statement point 4:**
  1. In Section 2.4, 2.5 and 2.6, different security testing tools were described and evaluated, in which Acunetix WVS was selected. Furthermore, in Section 2.5.1, the author justified why Acunetix WVS was selected. With this, problem statement point 4a is fulfilled.
  2. In Section 4.3 the author explained which security tests were to be performed, and how the security testing process would be conducted. Furthermore, in Section 5.2 the author conducted the security testing of DAI and MAG. The security tests were carried out by three testers. As mentioned in Section 6.2, the number of testers was a side effect of the limited time and resources. Nevertheless, a security test containing four iterations (two iterations using the current methodology, and two iterations using the EAST methodology) was carried out. With this, problem statement point 4b is fulfilled.
  3. Based on the test results given in Section 5.3, an evaluation of the security testing methodologies has been made in Section 6.1. With this, problem statement point 4c is fulfilled.

### 6.3.2 Research goals

This section describes whether the author has attained the research goals defined in Section 1.3. This is done by associating the problem statement with the research goals, and thereby describing how the research goals are attained

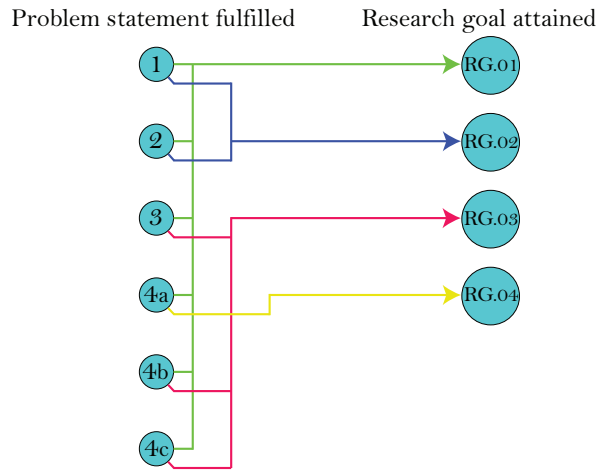


Figure 6.2: The left hand side of the figure shows the problem statements that are fulfilled. By fulfilling the problem statements, the research goals are attained, as shown on the right hand side of the figure.

as a result of fulfilling the problem statement (the fulfillment of the problem statement goals are described in Section 6.3.1). Figure 6.2 illustrates what is explained in the points below.

- **RG.01:** By fulfilling problem statement point 1, 2, 3 and 4, the author gained knowledge of state-of-the-art security testing methodologies, with a special focus on Web applications. With this, the author attained research goal RG.01.
- **RG.02:** By fulfilling problem statement point 1 and 2, the author found and evaluated security testing methodologies for Web applications. With this, the author attained research goal RG.02.
- **RG.03:** By fulfilling problem statement point 3, 4b and 4c, the author implemented one security testing methodology (at a proof of concept level) for Web applications into the SDLC applied by the AIS group at CERN, and evaluated it. With this, the author attained research goal RG.03.
- **RG.04:** By fulfilling problem statement point 4a, the author got an overview of the different security testing tool categories, along with some tool examples for each category (both freeware and commercial). With this, the author attained research goal RG.04.

## 6.4 The hypothesis: Verified or falsified?

The following hypothesis was defined by the author in Section 1.4:

**H.00** The detection of vulnerabilities in Web applications is done significantly more efficient regarding time spent, the amount of vulnerabilities that are found and managing false-positives by using a structured security testing methodology for Web applications, compared to existing ad hoc ways of performing security tests.

The author uses three factors as basis to decide whether H.00 is verified or falsified. As mentioned in Section 6.1, these factors are the efficiency of the security testing methodology regarding: (1) the amount of time spent, (2) the amount of vulnerabilities found and (3) the ability to mitigate false-positives. The following facts are derived from the results presented in Section 6.1:

- Equation (6.1) shows that the EAST methodology is approximately 21% more effective in average, regarding time spent, than the current security testing methodology.
- Equation (6.2) shows that the EAST methodology is approximately 95% more effective, regarding the amount of vulnerabilities found, than the current security testing methodology.
- The current security testing methodology does not have an activity in place to manage false-positives. It is therefore not possible to measure how efficient it is to manage false-positives using the EAST methodology, compared to managing false-positives using the current security testing methodology. Doing so, would be to compare “something” against “nothing”. However, only the EAST methodology provides the managing of false-positives, which further mitigates false-positives during the security testing process.

The three abovementioned points show that the EAST methodology (a structured security testing methodology for Web applications) is significantly more efficient than the current security testing methodology applied by the AIS group (an ad hoc way of performing security tests). *Hence, hypothesis H.00 is verified.*



# Chapter 7

## Conclusion and further work

This chapter highlights the main points and gives a discussion of the achievements. Furthermore, it presents thoughts and suggestions of potential future work and improvements.

### 7.1 Conclusion

One of the key security practices that needs to be in place in order to mitigate the increasing number of vulnerabilities in Web applications, is a structured security testing methodology. The nature of Web applications requires an iterative and evolutionary approach to development. Therefore, the structured security testing methodology needs to have the capability of being adapted to such an environment, and it needs to be specialized for Web applications.

The most applied security testing methodologies today are extensive and are sometimes too complicated with their many activities and phases. By applying such extensive security testing methodologies in the realm of Web applications, developers tend to neglect the testing process because the methodologies are considered to be; too time-consuming, lacking a significant payoff and inappropriate to be applied on Web applications because they have a very short time-to-market. This can be regarded as one of the factors to why security testing often is executed according to the penetrate-and-patch paradigm.

In this thesis, the author has shown that by using a structured security testing methodology especially developed for Web applications, leads to a significantly more effective way of performing security tests on Web applications compared to existing ad hoc ways of performing security tests. The factors that the author used to measure the efficiency were: **(1)** the amount of time spent on the security testing process, **(2)** the amount of vulnerabilities found during

the security testing process and **(3)** the ability to mitigate false-positives during the security testing process.

The author showed this by:

- Executing a research among state-of-the-art security testing methodologies for Web applications.
- Eliciting Agile Security Testing (based on predefined criterions) as the most adequate security testing methodology for Web applicaitons. Then, the author extended Agile Security Testing to make it support all the predefined criterions. The resulting methodology was named Extended Agile Security Testing (EAST) by the author.
- Eliciting a Web Vulnerability Scanner (a security testing tool) to be used in the EAST methodology.
- Integrating the EAST methodology into the SDLC (Scrum) applied by CERN's Administrative Information Services (AIS) group, at the General Infrastructure Services (GS) department.
- Performing security tests using the AIS group's security testing methodology (an ad hoc way of performing security tests), performing security tests using the EAST methodology (a structured security testing methodology for Web applications), and finally comparing the results obtained from the security tests.

The results of the last mentioned point showed that the EAST methodology is:

- Approximately 21% more effective in average, regarding time spent, than existing ad hoc ways of performing security tests.
- Approximately 95% more effective, regarding the amount of vulnerabilities found, than existing ad hoc ways of performing security tests.
- The only methodology that provides the managing of false-positives, which further mitigates false-positives during the security testing process.

## 7.2 Further work

Although the author has shown that the EAST methodology is significantly more efficient than existing ad hoc ways of performing security tests, future evaluations of the EAST methodology should be focused on mitigating the threats to the validity presented in Section 6.2. The following points list alternative questions that should be addressed to mitigate the threats to the validity:

- Does other tools produce significantly different results? Are the results better or worse?
- The security tests were performed by testers that had low security specific knowledge (the majority), and that had low security testing experience. Do testers that have medium or high security specific knowledge and security testing experience produce significantly improved test results?
- Is the EAST methodology still significantly more efficient than existing ad hoc ways of performing security tests when a larger empirical data is used? I.e. more testers, and more vulnerability classes? There were only three testers, and nine different vulnerability classes used in the test iterations in this thesis.

Besides the threats to the validity, future evaluations of the EAST methodology should also address the question of how efficient it is compared to extensive testing methodologies when applied on Web applications. The extensive security testing methodologies would naturally require more time and thereby be less efficient regarding time spent. However, what would be interesting to discover, is whether the EAST methodology lacks activities that are vital for the overall security testing process, compared to the extensive security testing methodologies.

Figure 4.4 shows that EAST step 4 (repository of knowledge) will also be carried out while the system is in operation. The author explained why it should be carried out. As a further work, it would be interesting to figure out *how* the repository of knowledge would be updated while the system is in operation.

Studying Table 5.10 and Table 5.11 reveals that the testers have spent most time (jointly) on EAST step 5 (penetration testing and mitigating false positives). A possible future improvement would be to find out why the testers spend most time on this step, and whether possible improvements of the step is possible.



# List of Figures

1.1	The usage of Web applications among US adults . . . . .	2
1.2	Reasons for not using Web applications . . . . .	2
1.3	Malicious code threats . . . . .	3
1.4	Research method and approach . . . . .	9
2.1	The Security Development Lifecycle (SDL) . . . . .	14
2.2	The security touchpoints . . . . .	14
2.3	The Secure Software Development Lifecycle (SSDL) . . . . .	15
2.4	The Open Source Security Testing Methodology Manual sections . . . . .	18
2.5	TDD: Test Driven Development . . . . .	19
2.6	The misuse case legend . . . . .	20
2.7	A misuse case example . . . . .	21
2.8	A highly testable architecture . . . . .	22
2.9	The V model . . . . .	23
2.10	Threat modeling . . . . .	25
2.11	The generic SDLC model . . . . .	28
2.12	A Security Goal Indicator Tree (SGIT) example . . . . .	31
2.13	A Vulnerability Inspection Diagram (VID) example . . . . .	32
2.14	Reviewing for cross site scripting (code review) . . . . .	33
2.15	The OWASP Testing Framework work flow . . . . .	35
2.16	Acunetix WVS scan result . . . . .	41
2.17	Acunetix WVS AcuSensor Technology . . . . .	42
2.18	Acunetix WVS Vulnerability Editor . . . . .	44
3.1	CERN's organizational structure . . . . .	56
3.2	LHC and its experiments . . . . .	57
3.3	The Scrum SDLC . . . . .	61
3.4	The current security testing methodology . . . . .	63

---

3.5	CORAS graphical modeling notation . . . . .	65
3.6	Risk analysis: Asset diagram . . . . .	67
3.7	Risk analysis: Threat diagram and risk estimations . . . . .	71
3.8	Risk analysis: Risk treatment . . . . .	72
4.1	Marking false-positives using Acunetix WVS . . . . .	76
4.2	Viewing false-positives in Acunetix WVS . . . . .	77
4.3	Extended Agile Security Testing . . . . .	78
4.4	Extended Agile Security Testing integrated in the AIS group's SDLC . . . . .	79
4.5	EDH screenshot . . . . .	83
4.6	Activity diagram of the testing process . . . . .	84
5.1	DAI screenshot . . . . .	94
5.2	MAG screenshot . . . . .	95
5.3	Misuse case diagram of DAI . . . . .	106
5.4	DAI folder and file structure . . . . .	107
5.5	PMD in action . . . . .	108
5.6	The Reflected XSS profile in Acunetix WVS . . . . .	110
5.7	DAI false-false positive tree . . . . .	110
6.1	Comparison of the test results . . . . .	119
6.2	Problem statement fulfilled, research goals attained . . . . .	123
A.1	XSS test - vulnerable Web application . . . . .	A-2
A.2	XSS test - XSS carried out . . . . .	A-2
A.3	Stored XSS test - input form . . . . .	A-3
A.4	Stored XSS test - Stored XSS carried out . . . . .	A-4
A.5	Code Information Leakage . . . . .	A-11

# List of Tables

1.1	Value of advertised goods in the underground economy . . . . .	3
1.2	Prevention is cheaper than cure . . . . .	5
1.3	Elements of Classical Research and Technology Research . . . . .	8
2.1	Security testing tools . . . . .	39
2.2	Security testing tools and SDLC phases . . . . .	39
2.3	Relevant competitors to Acunetix WVS . . . . .	48
2.4	Comparison of security testing methodologies for Web applications	53
3.1	Risk analysis: Asset table . . . . .	66
3.2	Risk analysis: High level risk table . . . . .	68
3.3	Risk analysis: Likelihood scale . . . . .	69
3.4	Risk analysis: Consequence scale . . . . .	69
3.5	Risk analysis: Risk matrix . . . . .	69
3.6	Risk analysis: Risk evaluation . . . . .	70
5.1	Vulnerabilities tested on DAI and MAG . . . . .	91
5.2	Test environment . . . . .	96
5.3	The testers . . . . .	97
5.4	PMD results of DAI . . . . .	107
5.5	DAI penetration testing results . . . . .	111
5.6	PMD results of MAG . . . . .	113
5.7	MAG penetration testing results . . . . .	114
5.8	DAI test results (current methodology) . . . . .	115
5.9	MAG test results (current methodology) . . . . .	115
5.10	DAI test time results (EAST methodology) . . . . .	116
5.11	MAG test time results (EAST methodology) . . . . .	116
5.12	Total vulnerabilities (EAST methodology) . . . . .	116





# Glossary

Artifact	A man-made object taken as a whole. In this context: E.g. a new algorithm for a computer program, a new SDLC, a new security testing methodology, etc., <a href="#">7</a>
ATAM	A Comprehensive Method for Architecture Evaluation, <a href="#">29</a>
Backlog items	In Scrum: a collection of refined backlog items derived from a Sprint Backlog, <a href="#">60</a>
Bug	A security vulnerability at the implementation level, <a href="#">4</a>
CORAS	A methodology for conducting security risk analysis, specially developed to support structured brainstorming for risk identification, risk estimation and risk treatment, <a href="#">1</a>
Exploratory testing	A collective term for simultaneous learning, test design, and test execution, <a href="#">24</a>
False-negative	An existing bug that is not detected by a testing tool, <a href="#">74</a>
False-positive	A nonexistent bug that is reported as detected by a testing tool, <a href="#">8</a>
Flaw	A security vulnerability at the design level, <a href="#">4</a>
Forum	A public meeting or assembly for open discussion, <a href="#">1</a>

---

IRC	Internet relay chat, <a href="#">1</a>
ISAR	The Independent Software Architecture Review methodology, <a href="#">29</a>
Product Backlog	In Scrum: An evolving, prioritized queue of business and technical functionality that needs to be developed into a system, <a href="#">60</a>
RMF	Risk Management Framework, <a href="#">1</a>
Scrum	An iterative incremental framework for managing complex work (such as new product development) commonly used with agile software development, <a href="#">10</a>
SDL	Security Development Lifecycle, <a href="#">1</a>
Security regression	Security is compromised due to a material change in the system. For example, a new hardware platform, a new release of a build or module, etc., <a href="#">23</a>
SNMP	Simple Network Management Protocol, <a href="#">41</a>
Sprint Backlog	In Scrum: One Product Backlog item, <a href="#">60</a>
SQL	Structured Query Language, <a href="#">1</a>
SSDL	Secure Software Development Lifecycle, <a href="#">1</a>
TDD	Test Driven Development, <a href="#">19</a>
UML	Unified Modeling Language, <a href="#">20</a>
Vulnerability class	A type of software vulnerability, e.g. SQL injection and Cross Site Scripting (XSS), <a href="#">4</a>
WVS	Web Vulnerability Scanner, <a href="#">13</a>

# Bibliography

- [1] About CERN. <http://public.web.cern.ch/public/en/about/Global-en.html> Last date accessed 2009-05-12.
- [2] Acunetix Web Vulnerability Scanner (WVS). <http://www.acunetix.com/vulnerability-scanner/> Last date accessed 2009-04-20.
- [3] Add N Edit Cookies (AnEC) 0.2.1.3. <https://addons.mozilla.org/en-US/firefox/addon/573> Last date accessed 2009-07-06.
- [4] AppDetective. <http://www.appsecinc.com/products/appdetective/index.shtml> Last date accessed 2009-04-20.
- [5] BinNavi. <http://www.zynamics.com/binnavi.html> Last date accessed 2009-04-21.
- [6] BugScam. <http://sourceforge.net/projects/bugscam> Last date accessed 2009-04-21.
- [7] BugTraq mailing list. <http://www.securityfocus.com/archive/1> Last date accessed 2009-03-21.
- [8] Burp Suite. <http://portswigger.net/suite/> Last date accessed 2009-04-20.
- [9] Cenzic Hailstorm Professional. <http://www.cenzic.com/products/cenzic-hailstormPro/> Last date accessed 2009-04-21.
- [10] Class Type Architecture: A Strategy for Layering Software Applications. <http://www.ambysoft.com/essays/classTypeArchitecture.html> Last date accessed 2009-03-17.
- [11] CLR Profiler. <http://msdn.microsoft.com/en-us/library/ms979205.aspx> Last date accessed 2009-04-21.

- 
- [12] Common Vulnerabilities and Exposures. <http://cve.mitre.org/> Last date accessed 2009-03-21.
- [13] Computer Emergency Readiness Team (CERT). <http://www.cert.org/> Last date accessed 2009-03-21.
- [14] Compuware BoundsChecker. <http://www.compuware.com/products/devpartner/studio.htm> Last date accessed 2009-04-21.
- [15] Convention for the establishment of a European organization for nuclear research. <http://dsu.web.cern.ch/dsu/ls/conventionE.htm> Last date accessed 2009-05-12.
- [16] CORAS editor v.2.0.b5. <http://coras.sourceforge.net/downloads.html> Last date accessed 2009-05-22.
- [17] Desaware CAS/Tester. <http://www.desaware.com/products/castester/index.aspx> Last date accessed 2009-04-21.
- [18] Fiddler. <http://www.fiddler2.com/fiddler2/> Last date accessed 2009-04-21.
- [19] FindBugs. <http://findbugs.sourceforge.net/> Last date accessed 2009-04-20.
- [20] Firefox Toolbar. <https://addons.mozilla.org/en-US/firefox/search?q=&cat=1%2C12> Last date accessed 2009-04-21.
- [21] Fortify Source Code Analyzer (SCA). [http://www.fortify.com/products/detect/in\\_development.jsp](http://www.fortify.com/products/detect/in_development.jsp) Last date accessed 2009-04-20.
- [22] Growth of web applications in the US. <http://rubiconconsulting.com/insight/whitepapers/2007/09/growth-of-web-applications-in.html> Last date accessed 2009-02-19.
- [23] HP WebInspect software. <http://www.spidynamics.com/products/webinspect/> Last date accessed 2009-04-20.
- [24] IBM Rational AppScan. <http://www-01.ibm.com/software/awdtools/appscan/> Last date accessed 2009-04-27.
- [25] IDA Pro. <http://www.datarescue.com/> Last date accessed 2009-04-21.
- [26] ISECOM. <http://www.isecom.org/> Last date accessed 2009-04-15.
- [27] JUnit. <http://www.junit.org/> Last date accessed 2009-04-21.

- 
- [28] Klocwork Insight. <http://www.klocwork.com/products/insight.asp>  
Last date accessed 2009-04-20.
- [29] Microsoft FxCop. <http://msdn.microsoft.com/en-us/library/bb429476.aspx> Last date accessed 2009-04-21.
- [30] Microsoft Security Configuration Tool set. <http://technet.microsoft.com/en-us/library/bb742512.aspx> Last date accessed 2009-04-21.
- [31] .NETMon. <http://www.foundstone.com/us/resources-free-tools.asp> Last date accessed 2009-04-21.
- [32] NeXpose. <http://www.rapid7.com/nexpose/overview.jsp> Last date accessed 2009-04-20.
- [33] OWASP. <http://www.owasp.org/> Last date accessed 2009-03-26.
- [34] OWASP Tools. [http://www.owasp.org/index.php/Category:OWASP\\_Tool](http://www.owasp.org/index.php/Category:OWASP_Tool) Last date accessed 2009-04-20.
- [35] OWASP Tools Project. [http://www.owasp.org/index.php/Category:OWASP\\_Tools\\_Project](http://www.owasp.org/index.php/Category:OWASP_Tools_Project) Last date accessed 2009-04-22.
- [36] OWASP Top 10 vulnerabilities. [http://www.owasp.org/index.php/Top\\_10\\_2007](http://www.owasp.org/index.php/Top_10_2007) Last date accessed 2009-06-10.
- [37] OWASP WebScarab. [http://www.owasp.org/index.php/Category:OWASP\\_WebScarab\\_Project](http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project) Last date accessed 2009-04-20.
- [38] Paros. <http://www.parosproxy.org/index.shtml> Last date accessed 2009-04-21.
- [39] PMD - Java source code scanner (Static Analysis Tool). <http://pmd.sourceforge.net/> Last date accessed 2009-07-03.
- [40] Rational PurifyPlus. <http://www-01.ibm.com/software/awdtools/purifyplus/> Last date accessed 2009-04-21.
- [41] Rough Auditing Tool for Security (RATS). <http://www.fortify.com/security-resources/rats.jsp> Last date accessed 2009-04-20.
- [42] Scuba. <http://www.imperva.com/products/scuba.html> Last date accessed 2009-04-20.
- [43] SeaMonster V3.0. <http://sourceforge.net/projects/seamonster/>  
Last date accessed 2009-07-03.

- [44] Security Innovation. <http://www.securityinnovation.com/> Last date accessed 2009-03-20.
- [45] Security tools. <http://sectools.org/> Last date accessed 2009-04-20.
- [46] SHIELDS - Detecting known security vulnerabilities from within design and development tools. <http://www.shields-project.eu/> Last date accessed 2009-04-16.
- [47] SiteDigger. <http://www.foundstone.com/us/resources-free-tools.asp> Last date accessed 2009-04-21.
- [48] SQLrecon. <http://www.specialopssecurity.com/labs/sqlrecon/1.0/down.php> Last date accessed 2009-04-20.
- [49] SSLDigger. <http://www.foundstone.com/us/resources-free-tools.asp> Last date accessed 2009-04-21.
- [50] Test Tool Evaluations. <http://www.uml.org.cn/Test/12/Automated%20Testing%20Tool%20Evaluation%20Matrix.pdf> Last date accessed 2009-04-20.
- [51] The Administrative Information Services group. <http://it-dep-ais.web.cern.ch/it-dep-AIS/mandate.asp> Last date accessed 2009-05-13.
- [52] The website of the world's first-ever Web server. <http://info.cern.ch/> Last date accessed 2009-05-13.
- [53] Visual Studio Team System. <http://msdn.microsoft.com/en-gb/teamsystem/default.aspx> Last date accessed 2009-04-21.
- [54] Acunetix. *Acunetix Web Vulnerability Scanner, Manual V6.0*. <http://www.acunetix.com/vulnerability-scanner/wvsmanual.pdf> Last date accessed 2009-04-24.
- [55] Ananta Security. *Web Vulnerability Scanners Evaluation*. <http://www.darknet.org.uk/content/files/WebVulnScanners.pdf> Last date accessed 2009-04-27.
- [56] M. Andrews. Guest Editor's Introduction The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.
- [57] Shanai Ardi, David Byers, Per Håkon Meland, Inger Anne Tondel, and Nahid Shahmehri. How can the developer benefit from security modeling? *Availability, Reliability and Security, International Conference on*, 0:1017–1025, 2007.

- [58] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.
- [59] Dejan Baca, Kai Petersen, Bengt Carlsson, and Lars Lundberg. Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter? *Availability, Reliability and Security, International Conference on, IEEE*, pages 804–810, 2009.
- [60] James Bach. Exploratory testing explained. *The Test Practitioner*, 2002.
- [61] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 2003.
- [62] A. Cockburn and J. Highsmith. Agile Software Development: The People Factor. *Computer*, 34(11), 2001.
- [63] M. Curphey, R. Arawo, and M.V. Foundstone. Web application security assessment tools. *IEEE Security & Privacy*, 4(4):32–41, 2006.
- [64] Heidi E. I. Dahl. The CORAS method for security risk analysis. In *Tutorial presentation at 7th Estonian Summer School on Computer and Systems Science in cooperation with the Nordic Network On DEpendable Systems (NODES), Otepää, Estonia.*, 2008.
- [65] Heidi E. I. Dahl, Ida Hogganvik, and Ketil Stølen. Structured semantics for the coras security risk modelling language. report stf07 a970. Technical report, SINTEF Information and Communication Technology, 2007.
- [66] T.B. Dao and E. Shibayama. Idea: Automatic Security Testing for Web Applications. In *Engineering Secure Software and Systems: First International Symposium, Essos 2009 Leuven, Belgium, February 4-6, 2009 Proceedings*, page 180. Springer-Verlag New York Inc, 2009.
- [67] G.A. Di Lucca, A.R. Fasolino, F. Faralli, and U. De Carlini. Testing Web applications. *Software Maintenance, 2002. Proceedings. International Conference on*, pages 310–319, 2002.
- [68] M. Dowd, J. McDonald, and J. Schuh. *The art of software security assessment: identifying and preventing software vulnerabilities*. Addison-Wesley, 2007.
- [69] E. Dustin, J. Rashka, and D. McDiarmid. *Quality Web Systems: Performance, Security, and Usability*. Addison-Wesley, 2002.
- [70] Gencer Erdogan and Egil Trygve Baadshaug. Extending SeaMonster to support vulnerability inspection modeling. Technical report, NTNU, Department of computer and information science, 2008.

- [71] European Commission - Information and Communication Technologies. *D1.2 Initial SHIELDS approach guide*. <http://er-projects.gf.liu.se/main.php/D1.2%20Initial%20SHIELDS%20approach%20guide.pdf?fileitem=1786183> Last date accessed 2009-04-16.
- [72] European Commission - Information and Communication Technologies. *D4.1 Initial specifications of the security methods and tools*. <http://er-projects.gf.liu.se/main.php/D4.1%20Initial%20specifications%20of%20the%20security%20methods%20and%20tools.pdf?fileitem=10551374> Last date accessed 2009-04-16.
- [73] Marc Fossi, Eric Johnson, Dean Turner, Trevor Mack, Joseph Blackbird, David McKinney, Mo King Low, Téo Adams, Marika Pauls Laucht, and Jesse Gough. Symantec Report on the Underground Economy: July 2007 - June 2008. Technical report, Symantec Corporation, 2008.
- [74] P. Godefroid, M.Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.
- [75] A.R. Hevner, S.T. March, J. Park, and S. Ram. Design science in information systems research. *Management Information Systems Quarterly*, 28(1):75–106, 2004.
- [76] Hewlett-Packard. *HP WebInspect User Guide*. [https://products.spidynamics.com/webinspect/webinspectuserguide\\_8.0.548.0\\_04012009.pdf](https://products.spidynamics.com/webinspect/webinspectuserguide_8.0.548.0_04012009.pdf) Last date accessed 2009-02-29.
- [77] Edward Hieatt and Robert Mee. Going Faster: Testing The Web Application. *IEEE Software*, 19(2):60–65, 2002.
- [78] J. Highsmith and A. Cockburn. Agile Software Development: The Business of Innovation. *Computer*, 34(9):120–122, 2001.
- [79] Ida Hogganvik. *A graphical approach to security risk analysis*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2007.
- [80] Paco Hope and Ben Walther. *Web Security Testing Cookbook*. O'Reilly, 2008.
- [81] M.A. Howard. A process for performing security code reviews. *Security & Privacy, IEEE*, 4(4):74–79, 2006.
- [82] Michael Howard and Steve Lipner. *The Security Development Lifecycle: SDL, a Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.



- [83] IBM. *IBM Rational AppScan*. [ftp://ftp.software.ibm.com/software/rational/web/datasheets/watchfire\\_appscan\\_ds.pdf](ftp://ftp.software.ibm.com/software/rational/web/datasheets/watchfire_appscan_ds.pdf) Last date accessed 2009-02-22.
- [84] Institute for Security and Open Methodologies (ISECOM). *Open Source Security Testing Methodology Manual V2.2*. <http://www.isecom.org/mirror/osstmm.en.2.2.zip> Last date accessed 2009-04-15.
- [85] M. Jazayeri. Some trends in web application development. In *International Conference on Software Engineering*, pages 199–213. IEEE Computer Society Washington, DC, USA, 2007.
- [86] J. Jürjens. Model-based Security Testing Using UMLsec. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 220(1):93–104, 2008.
- [87] V. Kongsli. Towards agile security in web applications. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 805–808. ACM Press New York, NY, USA, 2006.
- [88] Tim Koomen, Leo van der Aalst, Bart Broekman, and Michiel Vroon. *TMap Next: For Result-driven Testing*. UTN Publishers, 2006.
- [89] S. Lipner and M. Howard. The Trustworthy Computing Security Development Lifecycle. In *20th Annual Computer Security Applications Conference (ACSAC 2004)*, pages 2–13, 2004.
- [90] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing Web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.
- [91] J.F. Maranzano, S.A. Rozsypal, G.H. Zimmerman, G.W. Warnken, P.E. Wirth, and D.M. Weiss. Architecture reviews: practice and experience. *Software, IEEE*, 22(2):34–43, 2005.
- [92] S.T. March and G.F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, 1995.
- [93] G. McGraw. Software security. *Security & Privacy Magazine, IEEE*, 2(2):80–83, 2004.
- [94] G. McGraw. Automated Code Review Tools for Security. *Computer*, 41(12):108–111, 2008.
- [95] Gary McGraw. *Software Security: Building Security in*. Addison-Wesley, 2006.

- [96] Gary McGraw and Bruce Potter. Software Security Testing. *IEEE Security & Privacy*, 2(5):81–85, 2004.
- [97] Microsoft. *Microsoft Security Development Lifecycle (SDL) V3.2*. <http://www.microsoft.com/downloads/details.aspx?FamilyID=2412c443-27f6-4aac-9883-f55ba5b01814&displaylang=en> Last date accessed 2009-04-02.
- [98] Microsoft. *The Microsoft Security Development Lifecycle (SDL): Process Guidance*. <http://msdn.microsoft.com/en-us/security/cc420639.aspx> Last date accessed 2009-03-21.
- [99] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The art of software testing*. John Wiley and Sons, 2004.
- [100] N-Stalker. *N-Stalker user manual*. <http://community.nstalker.com/manual/> Last date accessed 2009-02-22.
- [101] National Academy of Engineering, National Academy of Engineering of the N, and Microsoft Corporation. *Frontiers of engineering: reports on leading-edge engineering from the 2007 symposium*. National Academies Press, 2008.
- [102] The Open Web Application Security Project. *OWASP Testing Guide V3.0*. [http://www.owasp.org/index.php/Category:OWASP\\_Testing\\_Project](http://www.owasp.org/index.php/Category:OWASP_Testing_Project) Last date accessed 2009-03-26.
- [103] J. Peeters. Agile Security Requirements Engineering. In *Symposium on Requirements Engineering for Information Security*, 2005.
- [104] W. Radosevich, C. C. Michael, and Inc. Cigital. Black box security testing tools. <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261-BSI.html> Last date accessed 2009-04-17.
- [105] L. Røstad. An extended misuse case notation: Including vulnerabilities and the insider threat. In *The Twelfth Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ.06)*, 2006.
- [106] I. Rus and M. Lindvall. Knowledge management in software engineering. *Software, IEEE*, 19(3):26–38, 2002.
- [107] P.A.P. Salas, Padmanabhan Krishnan, and K.J. Ross. Model-Based Security Vulnerability Testing. *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, 0:284–296, 2007.

- [108] K. Schwaber. Scrum development process. In *OOPSLA Business Object Design and Implementation Workshop*, 1995.
- [109] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.
- [110] G. Sindre and A.L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
- [111] SINTEF, Telenor. *The CORAS UML Profile*. <http://coras.sourceforge.net/> Last date accessed 2009-02-22.
- [112] Ketil Stølen and Ida Solheim. Technology research explained. report sintef a313. Technical report, SINTEF Information and Communication Technology, 2007.
- [113] A. Tang, F.C. Kuo, and M.F. Lau. Towards Independent Software Architecture Review. In *Proceedings of the 2nd European conference on Software Architecture*, pages 306–313. Springer, 2008.
- [114] A. Tappenden, P. Beatty, J. Miller, A. Geras, and M. Smith. Agile security testing of Web-based systems via HTTPUnit. *Agile Conference, 2005. Proceedings*, pages 29–38, 2005.
- [115] Herbert H. Thompson. Why Security Testing Is Hard. *IEEE Security and Privacy*, 1(4):83–86, 2003.
- [116] Herbert H. Thompson. Application penetration testing. *IEEE Security and Privacy*, 3(1):66–69, 2005.
- [117] G. Tóth, G. Kőszegi, and Z. Hornák. Case study: automated security testing on the trusted computing platform. In *Proceedings of the 1st European workshop on system security*, pages 35–39. ACM New York, NY, USA, 2008.
- [118] Dean Turner, Marc Fossi, Eric Johnson, Trevor Mack, Joseph Blackbird, Stephen Entwisle, Mo King Low, David McKinney, and Candid Wueest. Symantec Internet Security Threat Report: Trends for July-December 07 - Volume XIII. Technical report, Symantec Corporation, 2008.
- [119] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2002.
- [120] James A. Whittaker and Herbert H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing*. Pearson/Addison Wesley, 2003.

- [121] Chris Wysopal, Luke Nelson, Elfriede Dustin, Lucas Nelson, and Dino Dai Zovi. *The Art of Software Security Testing*. Addison-Wesley, 2006.

# Appendices

# Appendix A

## The current methodology: the guidelines

The following points give guidelines for security testers to perform security tests of the vulnerabilities that are given in Table 5.1. These guidelines are what the AIS group use for the current security testing methodology. Some of the guidelines have been adapted from the OWASP Testing Guide V3.0 [102] and the OWASP Top 10 vulnerabilities Web site [36]. The guidelines also give a short description of the underlying vulnerability followed by a detailed description. The detailed descriptions are omitted in the points below as they are not needed in order to perform the security test. Furthermore, the terms “application” and “Web application” are used interchangeably in the points below.

### VT.01 - Reflected XSS

- **Short description:** Reflected Cross Site Scripting (XSS) is another name for non-persistent XSS, where the attack doesn't load with the vulnerable Web application but is originated by the victim loading the offending URI.
- **How to test:**
  1. Detect input vectors. The tester must determine the Web application's variables and how to input them in the Web application.
  2. Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but trigger responses from the Web browser that



Figure A.1: A Web application potentially vulnerable to XSS. This figure is adapted from OWASP Testing Guide V3.0 [102].

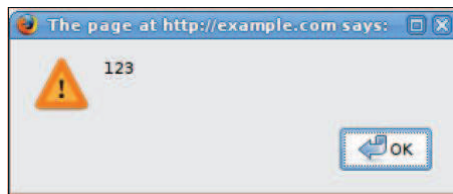


Figure A.2: XSS successfully executed. This figure is adapted from OWASP Testing Guide V3.0 [102].

manifests the vulnerability. Testing data can be generated by using a Web application fuzzer or manually.

3. For each vulnerability reported in the previous phase, the tester will analyze the report and attempt to exploit it with an attack that has a realistic impact on the Web application's security.

- **Example:** Consider a site that has a welcome notice “Welcome Username”, where “Username” denotes the user's name. Further, let's say that the user's name is MrSmith. The URI:

```
http://example.com/index.php?user=MrSmith
```

would give the result as shown in Figure A.1. The tester must suspect that every data entry point can result in an XSS attack. By changing the URI to:

```
http://example.com/index.php?user=<script>alert(123)</script>
```

and given that no input validation for that specific URI is done, then the result would be as shown in Figure A.2. If this happens, then there is an indication that the Web application is vulnerable to Reflected Cross Site Scripting.

User Details	
Name:	Administrator
Username:	admin
Email:	aaa@aa.com
New Password:	
Verify Password:	

Figure A.3: A Web application form is one of the places where Stored XSS can be performed. This figure is adapted from OWASP Testing Guide V3.0 [102].

## VT.02 - Stored XSS

- **Short description:** Stored Cross Site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack.
- **How to test:** The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:
  - User/Profiles page: the application allows the user to edit/change profile details such as first name, last name, nickname, avatar, picture, address, e-mail address etc.
  - Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later.
  - File Manager: application that allows upload of files.
  - Application settings/preferences: application that allows the user to set preferences.
  - Forums: application that allows storing of user defined text.

The second step is to analyze the HTML code. Input stored by the application is normally used in HTML tags, but it can also be found as part of Javascript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. For example, the form in Figure A.3 has five input fields, in which one of them is an e-mail address. The HTML code for the Email input field is: `<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />`. In this case, the penetration tester needs to find a way to inject code outside the `<input>` tag.

- **Example:** Below are two basic injection examples:

```
aaa@aa.com"><script>alert(document.cookie)</script>
```



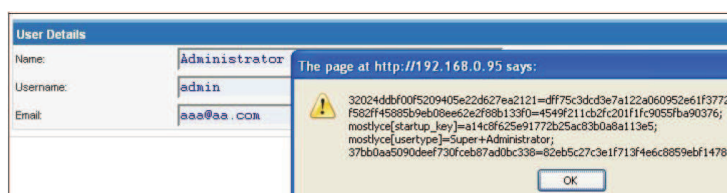


Figure A.4: Stored XSS successfully executed. This figure is adapted from OWASP Testing Guide V3.0 [102].

```
aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E
```

Input validation/filtering controls of the application are tested by entering these injection examples in the Email input field and submitting the form. The tester needs to ensure that the input is submitted through the application. This normally involves disabling Javascript if client-side security controls are implemented or modifying the HTTP request with a Web proxy (e.g. WebScarab). If one of the abovementioned injection examples are successful, then the result would be a popup window containing the cookie values as shown in Figure A.4. This would happen each time the browser reloads the page. The resulting HTML code after the injection is: `<input class="inputbox" type="text" name="email" size="40" value="'aaa@aa.com'><script>alert(document.cookie)</script>`

### VT.03 - SQL Injection

- **Short description:** A SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.
- **How to test:** The first step in this test is to understand when the application connects to a DB Server in order to access some data. Typical examples of cases when an application needs to execute transactions on a DB include:

- Authentication forms: when authentication is performed using a Web

form, chances are that the user credentials are checked against a database that contains all usernames and passwords.

- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- E-Commerce sites: the products and their characteristics (price, description, availability, etc.) are very likely to be stored in a relational database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. The very first test usually consists of adding a single quote (') or a semicolon (;) to the field under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The error messages are valuable to the tester in order to perform a successful SQL injection.

- **Example:** The following SQL query is typically used from the Web application in order to authenticate a user:

```
SELECT * FROM Users WHERE Username='$username' AND
Password='$password'
```

If the query returns a value it means that inside the database a user with that credentials exists, then the user is allowed to login to the system, otherwise the access is denied. The values of the input fields are generally obtained from the user through a Web form. Suppose the following Username and Password values are inserted:

```
$username = 1' or '1' = '1
$password = 1' or '1' = '1
```

Then the SQL query will be:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1'
OR '1' = '1'
```

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable Web site

is `www.example.com`, the request that is carried out would be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1
&password=1'%20or%20'1'%20=%20'1
```

After a short analysis it is possible to notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password. Another example is the following query:

```
SELECT * FROM Users WHERE ((Username='$username') AND
>Password=MD5('$password'))
```

There are two problems in this case, one is the use of the parentheses, and the second problem is the use of the MD5 hash function. To resolve the problem of the parentheses a number of closing parentheses (until we obtain a corrected query) is added. The second problem is resolved by invalidating the part of the query that contains the MD5 hash function. To invalidate the MD5 hash function the symbol for a comment is typed in. The comment symbol can vary for each database systems. In Oracle the symbol is “--”. Given this information the values for user name and password is set to:

```
$username = 1' or '1' = '1'))/*
$password = foo
```

The resulting SQL query will then become:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND
>Password=MD5('$password'))
```

The URL request will then be:

```
http://www.example.com/index.php?username=1'%20or%20'1'
%20=%20'1'))/*&password=foo
```

#### VT.04 - Malicious File Execution

- **Short description:** Malicious file execution can be carried out if the underlying Web application doesn't check whether the uploaded file is what

the file is said to be. E.g. a file that has the extension .jpg may actually be a file that contains a script, which may further be executed by the Web application each time the file is invoked. All Web application frameworks are vulnerable to malicious file execution if they accept filenames or files from the user.

- **How to test:** If the Web application allows to upload a file, the tester has to create a file containing a script and then upload it:
  1. Create a script in the programming/scripting language the Web application supports, or is supposed to support.
  2. Upload the file containing the script with its original file extension e.g. .exe, .php, .js or .sh.
  3. If the application doesn't accept the file, try to change the extension and upload again.
  4. If the application now accepts the file, try to invoke the file and see if the script is executed, if it is executed, then the application is vulnerable to malicious file execution.
- **Example:** At the simplest, the following Javascript code can be written in a file and uploaded:

```
<script type='text/javascript'>  
alert(document.cookie);  
</script>
```

Whenever the file that contains the abovementioned script is invoked, the cookie value for that specific user is shown in a popup dialog.

#### VT.05 - Insecure Direct Object Reference

- **Short description:** A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. An attacker can manipulate direct object references to access other objects without authorization, unless an access control check is in place.
- **How to test:** There are two ways to test for this vulnerability. The first option is to manipulate the values that are available in the URL field in the browser. The second option is to download the HTML source from the application Web site to a local computer, and then manipulate the

hidden fields. The following points show how to test for insecure direct object reference in a URL:

1. Let's say we have the following URL:  
`http://www.example-bank.com/BankAccount?AccountNumber=01234567890.`
2. By changing the AccountNumber variable in the URL, information about other bank accounts can be visible if no proper access control check is in place.

As long as there is a variable in the URL that may be changed, a test of this type is possible. If information that should have been restricted is accessed or displayed, then an insecure direct object reference attack has been successfully carried out. The following points show how to test for insecure direct object reference in the HTML source code of a Web application:

1. Download the HTML source code for the Web application to a local computer. This can be done by clicking on View and then clicking on Source via Microsoft Internet Explorer. Other browsers have similar functionality.
2. Look for the `<input>` HTML tag where type is set to be hidden, e.g. `<input type='hidden' name='AccountNumber' />`.
3. Modify `type='hidden'` to `type='text'`.
4. In the form's `<form>` tag, modify the `action` attribute to point directly to the issuing Web site. E.g. in this example the complete `<form>` tag would be:

```
<form method='post' action='http://www.example-bank.com/'  
name='MainForm'>
```

This makes sure that the form would be submitted to the issuing Web application.

5. Save the file as a HTML file on the local computer and open it using a browser.
6. If everything went well, the hidden variables should be visible and it should be possible to type in any value in the AccountNumber input field.
7. Finally, it should be possible to see the response from the issuing Web application by submitting the form.

- **Example:** Look at the previous point for an example.

**VT.06 - Cross Site Request Forgery (CSRF)**

- **Short description:** CSRF is an attack which forces an end user to execute unwanted actions on a Web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may force the users of a Web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation, when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire Web application.
- **How to test:** The tester needs to know URLs in the restricted (authenticated) area. If the tester possess valid credentials, he/she can assume both roles—the attacker and the victim. In this case, the tester know the URLs to be tested just by browsing around the application.

Otherwise, if the tester doesn't have valid credentials available, he/she has to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link. This may involve a substantial level of social engineering.

Either way, a test case can be constructed as follows:

- Let  $u$  denote the URL being tested; for example,  $u = \text{http://www.example.com/action}$ .
  - Build an HTML page containing the HTTP request referencing URL  $u$  (specifying all relevant parameters; in the case of HTTP GET this is straightforward, while to a POST request you need to resort to some Javascript).
  - Make sure that the valid user is logged on the application.
  - Induce him into following the link pointing to the to-be-tested URL (social engineering involved if you cannot impersonate the user yourself).
  - Observe the result, i.e. check if the Web server executed the request.
- **Example:** Look at the previous point for an example.

**VT.07 - Information Leakage and Improper Error Handling**

- **Short description:** Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Applications can also leak internal state via how long they take to process certain operations or via different responses to differing inputs, such as displaying the same error text with different error numbers. Web applications will often leak information about

their internal state through detailed or debug error messages. Detailed or debug error messages are good entry points, and are often used to launch or even automate more powerful attacks.

- **How to test:** The basic purpose of this vulnerability testing class is to make the application produce/leak system information (as described in the previous point). This is usually done in the following way:
  - From a Graphical User Interface (GUI) perspective, a tester can type unvalid information in e.g. forms in order to crash the application. If there are no tests in the Web application to check whether unexpected values are submitted to the Web application, the Web application may crash and reveal internal error messages.
  - From a developer (programmer) perspective, a tester can modify source code in order to make the application crash and see if the application reveals e.g. the error stack trace on the browser.
- **Example:** Figure A.5 shows an example of code information leakage which is due to a NullPointerException in the Web application. The correct way to prevent this is to send the information that is shown in the figure to a log file, and to use a try—catch statement in the source code in order to catch the NullPointerException. Furthermore, an error message indicating that an error occurred has to be shown to the user. The error message needs only to contain e.g. “An error occurred, please try again later”. A typical example of improper error handling is when a Web application gives “incorrect password” as an error message when a user types in correct username but incorrect password. Such information is valuable for a malicious user. If a malicious user knows that the username is correct, then he/she can perform e.g. a dictionary attack in order to guess the correct password.

#### VT.08 - Broken Authentication and Session Management

- **Short description:** Proper authentication and session management is critical to Web application security. Flaws in this area most frequently involve the failure to protect credentials and session tokens through their lifecycle. These flaws can lead to the hijacking of user or administrative accounts, undermine authorization and accountability controls, and cause privacy violations.
- **How to test:** All interaction between the client and application should be tested at least against the following criteria:

**message**

**description** The server encountered an internal error () that prevented it from fulfilling this request.

**exception**

```
org.apache.jasper.JasperException: Exception in JSP: /ManageAccountProfile.jsp:71

68:         {/if oldPassword field is not empty, but question, answer, password and confirmPassword fields
69:         //are empty show error messages and set oldPasswordCheck = false;
70:         //answerCheck = false; questionCheck = false;
71:         if (!request.getParameter("oldPassword").trim().equals("")) &&
72:         request.getParameter("question").trim().equals("") &&
73:         request.getParameter("answer").trim().equals("") &&
74:         request.getParameter("password").trim().equals("") &&
```

**Stacktrace:**

```
org.apache.jasper.servlet.JspServletWrapper.handleJspException(JspServletWrapper.java:451)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:373)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:329)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265)
javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
```

**root cause**

```
java.lang.NullPointerException
org.apache.jsp.ManageAccountProfile_jsp._jspService(ManageAccountProfile_jsp.java:139)
org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java:98)
javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
org.apache.jasper.servlet.JspServletWrapper.service(JspServletWrapper.java:331)
org.apache.jasper.servlet.JspServlet.serviceJspFile(JspServlet.java:329)
org.apache.jasper.servlet.JspServlet.service(JspServlet.java:265)
javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
```

**note** The full stack trace of the root cause is available in the Apache Tomcat/5.5.27 logs.

**Apache Tomcat/5.5.27**

Figure A.5: Due to a `NullPointerException` in the source code, the error message that can be seen on the figure is produced by Apache Tomcat Web server. The problem is not Apache Tomcat, but improper error handling in the application. From this error message it is possible to deduce facts such as the name of the Java servlet that crashed, and the values of the HTML form input-fields.



- Are all Set-Cookie directives tagged as secure?
- Do any Cookie operations take place over unencrypted transport?
- Can the Cookie be forced over unencrypted transport?
- If so, how does the application maintain security?
- Are any Cookies persistent?
- What Expires= times are used on persistent cookies, and are they reasonable?
- Are cookies that are expected to be transient configured as such?
- What HTTP/1.1 Cache-Control settings are used to protect Cookies?
- What HTTP/1.0 Cache-Control settings are used to protect Cookies?

The first step required in order to manipulate the cookie is to understand how the application creates and manages cookies. This can be done by providing answers to the following questions:

- How many cookies are used by the application?

Surf the application. Note when cookies are created. Make a list of received cookies, the page that sets them (with the setcookie directive), the domain for which they are valid, their value, and their characteristics.

- Which parts of the application generate and/or modify the cookie?

Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie?

- Which parts of the application require this cookie in order to be accessed and utilized?

Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.

Analyzing the information gathered from the abovementioned points gives enough basis to test for session ID predictability, cookie reverse engineering, brute force attacks through cookie manipulation, and testing for how well sessions are managed.

- **Example:** A test that checks whether the sessions are managed properly can be performed by carrying out the following points:

1. Login to a Web application using valid credentials.

2. Use the browser's cookie management tool (or a similar tool) to obtain the session ID given by the Web application.
3. Log out of the Web application and use the cookie management tool to delete the cookie and its information.
4. Open the Web site to the same Web application in step 1, and login again using valid credentials.
5. Log out of the Web application.
6. Use the cookiemanager to edit the new cookie and delete the new session ID. Then paste the old session ID in the session ID field.
7. Try to navigate in the Web application.

If this test is successfully carried out, then the tester would be able to navigate in the Web application using the old session ID. This indicates e.g. that the unused sessions are not timed out. Furthermore, two testers (tester A and tester B) may perform the test in the following way:

1. Tester A performs step 1–3 using credentials A.
2. Tester B obtains tester A's session ID.
3. Tester B performs step 4 using credentials B.
4. Tester B performs step 5.
5. Tester B performs step 6 and replaces the new session ID with the session ID obtained from tester A.
6. Tester B performs step 7 and now navigates the Web application as tester A. This is also referred to as session hijacking.

#### **VT.09 - Failure to Restrict URL Access**

- **Short description:** Frequently, the only protection for a URL is that links to that page are not presented to unauthorized users. However, a motivated, skilled, or just plain lucky attacker may be able to find and access these pages, invoke functions, and view data. Security by obscurity is not sufficient to protect sensitive functions and data in an application. Access control checks must be performed before a request to a sensitive function is granted, which ensures that the user is authorized to access that function.
- **How to test:** The primary attack method for this vulnerability is called "forced browsing", which encompasses guessing links and brute force techniques to find unprotected pages. Before trying to guess folder names or file names, a site structure should be obtained at least by carrying out the following points:

1. Browses the Web site and observe which link points to which folder/file.
  2. During the browsing, create a list of links and which folders/files they point to. This is to get an overview of the site structure.
  3. Try to look for a pattern of the folder and file names by analyzing the result.
  4. Make educated guesses from the knowledge obtained by the site structure.
- **Example:** Some typical examples are `/admin/adduser.php` and `/approveTransfer.do`. Otherwise, as explained in the previous point, to test for this vulnerability class is to conduct “forced browsing”.