Evaluation of the applicability of security testing techniques in continuous integration environments

by

Pontus Thulin

LIU-IDA/LITH-EX-A--14/063--SE

2015-01-19
Master’s Thesis

Evaluation of the applicability of security testing techniques in continuous integration environments

by

Pontus Thulin

LIU-IDA/LITH-EX-A--14/063--SE

2015-01-19

Supervisors:  
Ulf Kargén (IDA, Linköpings Universitet)  
Sebastian Åkerman (Omegapoint AB)

Examiner:  
Nahid Shahmehri (IDA, Linköpings Universitet)
Abstract

Agile development methodologies are becoming increasingly popular, especially in projects that develop web applications. However, incorporation of software security in lightweight approaches can be difficult. Using security testing techniques throughout a complete agile development process by running automated tests in continuous integration environments is one approach that strives to improve security in agile projects. Instead of performing security testing at the end of the development cycle, such methods enable early and continuous detection of security risks and vulnerabilities.

The purpose of this thesis is to study how existing security testing techniques operate in continuous integration environments and what level of security they can help assure. The work is a qualitative analysis of different security testing techniques and evaluates how they technically fit into a continuous integration environment as well as how they adhere to agile principles. These techniques are also analyzed with the use of OWASP Top Ten to determine which security requirements they can verify. The outcome of the analysis is that no existing security testing technique is a perfect fit for usage in continuous integration testing. Each technique has its distinct advantages and drawbacks that should be taken into consideration when choosing a technique to work with in continuous integration environments.
## Contents

1 Introduction ................................................. 1
   1.1 Motivation and Background ......................... 1
   1.2 Purpose ........................................... 2
   1.3 Framing the Problem ................................ 3
   1.4 Scope ............................................ 3
   1.5 Method ........................................... 4
   1.6 Outline .......................................... 4
   1.7 Related Work ....................................... 6
   1.8 Evaluation of Sources ............................... 6
   1.9 Acknowledgements .................................. 7

2 Security Risks and Requirements ......................... 8
   2.1 Functional Security Requirements ................ 8
   2.2 Non-Functional Security Requirements ............. 9
   2.3 Requirement verification ............................ 9
   2.4 Security Risks .................................... 10
   2.5 OWASP Top Ten .................................... 12
     2.5.1 A1 - Injection ................................ 13
     2.5.2 A2 - Broken Authentication and Session Management ... 13
     2.5.3 A3 - Cross-site Scripting (XSS) ................. 14
     2.5.4 A4 - Insecure Direct Object Reference .......... 14
     2.5.5 A5 - Security Misconfiguration ................. 14
     2.5.6 A6 - Sensitive Data Exposure .................. 14
     2.5.7 A7 - Missing function level access control .... 15
     2.5.8 A8 - Cross-site Request Forgery (CSRF) ........ 15
     2.5.9 A9 - Using components with known vulnerabilities 15
     2.5.10 A10 - Unvalidated Redirects and Forwards .... 15

3 Security in Agile Software Development Processes ........ 16
   3.1 Software Development Processes .................. 16
   3.2 The Agile Manifesto ................................ 17
   3.3 Software Security ................................. 18
   3.4 Secure Development Life cycle ...................... 18
   3.5 Software Security in Agile ......................... 18
   3.6 Continuous Integration .............................. 19
     3.6.1 Security Testing in Continuous Integration .... 20
3.7 Wanted properties in Continuous Integration .......................... 22
   3.7.1 Automated execution ........................................... 22
   3.7.2 Headless .......................................................... 22
   3.7.3 Repeatable ........................................................ 22
   3.7.4 Internal expertise ............................................... 23
   3.7.5 Automated decisions .......................................... 23
   3.7.6 Fast execution ............................................... 24
   3.7.7 Early testing ................................................... 25
   3.7.8 Simple integration ............................................. 25

4 Security Testing Techniques .................................... 26
   4.1 Vulnerabilities, Bugs and Flaws ................................. 26
      4.1.1 Common Design Issues .................................... 27
      4.1.2 Common Implementation Issues ............................ 28
      4.1.3 Security Metrics .......................................... 29
   4.2 Application Security Testing Categories ....................... 30
   4.3 Static Application Security Testing ............................. 31
      4.3.1 Testing Scope .............................................. 32
      4.3.2 Underlying Static Analysis Techniques .................... 33
      4.3.3 Detection of Vulnerabilities ............................... 35
      4.3.4 False Positives vs. False Negatives ...................... 36
   4.4 Dynamic Application Security Testing ......................... 37
      4.4.1 Testing Scope .............................................. 37
      4.4.2 Penetration Testing ....................................... 38
   4.5 Interactive Application Security Testing ...................... 44
      4.5.1 Testing Scope .............................................. 44
      4.5.2 Runtime analysis .......................................... 45
      4.5.3 Combination of DAST and Runtime analysis ............... 47
      4.5.4 Combination of SAST and DAST ............................ 48

5 Analysis ...................................................... 49
   5.1 Evaluation of AST in Continuous Integration .................. 49
      5.1.1 Automated execution .................................... 50
      5.1.2 Headless .................................................. 51
      5.1.3 Repeatable ................................................ 52
      5.1.4 Internal expertise ........................................ 53
      5.1.5 Automated decisions ..................................... 55
      5.1.6 Fast execution ........................................... 56
      5.1.7 Early testing .............................................. 58
      5.1.8 Simple integration ...................................... 59
      5.1.9 Property evaluation summary ............................. 60
   5.2 Evaluation of Security Requirement Verification using AST ... 61
      5.2.1 A1 - Injection .......................................... 61
      5.2.2 A2 - Broken Authentication and Session Management .... 63
      5.2.3 A3 - Cross-site Scripting (XSS) ......................... 64
      5.2.4 A4 - Insecure Direct Object Reference ................... 65
      5.2.5 A5 - Security Misconfiguration ........................... 66
      5.2.6 A6 - Sensitive Data Exposure ........................... 66
<table>
<thead>
<tr>
<th>5.2.7</th>
<th>A7 - Missing function level access control</th>
<th>67</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.8</td>
<td>A8 - Cross-site Request Forgery (CSRF)</td>
<td>68</td>
</tr>
<tr>
<td>5.2.9</td>
<td>A9 - Using components with known vulnerabilities</td>
<td>69</td>
</tr>
<tr>
<td>5.2.10</td>
<td>A10 - Unvalidated Redirects and Forwards</td>
<td>70</td>
</tr>
<tr>
<td>5.2.11</td>
<td>Summary of evaluation of Security Requirement Verification</td>
<td>73</td>
</tr>
</tbody>
</table>

### 6 Conclusion

<table>
<thead>
<tr>
<th>6.1</th>
<th>Discussion</th>
<th>75</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Contributions</td>
<td>77</td>
</tr>
<tr>
<td>6.3</td>
<td>Recommendations and Future work</td>
<td>78</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Introducing, finding and repairing defects\[35\] ........................................ 2
1.2 Method ......................................................... 5

2.1 Verification of security requirements ........................................ 10
2.2 Risks as paths through a system\[18\] ........................................ 11
2.3 Security risk ......................................................... 12

3.1 Illustration of an agile development process\[2\] ................................. 17
3.2 Illustration of a Continuous Integration environment\[48\] ......................... 19
3.3 Illustration of a Continuous Delivery chain\[6\] ................................ 20
3.4 Illustration of security testing in CI ........................................ 21
3.5 Sequential illustration of security testing in CI\[51\] ............................... 21

4.1 Vulnerabilities as series of events\[29\] ........................................ 27
4.2 Automated decisions using metrics ........................................ 30
4.3 Illustration of SAST’s scope of analysis ........................................ 32
4.4 An illustration of a control flow graph\[8\] ........................................ 34
4.5 Combination of static analysis techniques ........................................ 35
4.6 False positives and negatives ......................................................... 37
4.7 Dynamic analysis tests from the outside ........................................ 37
4.8 Testing scope of DAST ......................................................... 38
4.9 Illustration of techniques used in penetration testing ........................... 39
4.10 Comparison of testing time for various DAST tools\[31\] ............................. 40
4.11 Comparison of network traffic used for various DAST tools\[31\] ................. 40
4.12 SSL with ZAP\[61\] ......................................................... 42
4.13 Possible test coverage with IAST ........................................ 45
4.14 Runtime analysis using sensors\[59\] ........................................ 46
4.15 Runtime analysis tracks data by tagging it\[29\] ................................ 46
4.16 Combination of DAST and Runtime Analysis\[59\] ................................ 47
List of Tables

5.1 Estimated score of the automated execution property .................. 51
5.2 Estimated score of the headless property .............................. 52
5.3 Estimated score of the repeatable property ............................ 53
5.4 Estimated score of the internal expertise property .................... 55
5.5 Estimated score of the automated decisions property ................. 56
5.6 Estimated score of the fast execution property ....................... 57
5.7 Estimated score of the early testing property ......................... 59
5.8 Estimated score of the simple integration property .................. 60
5.9 Estimated score of all properties .......................... 61
5.10 Estimated coverage of A1 - Injection .................................. 63
5.11 Estimated coverage of A2 - Broken Authentication and Session Management ............................. 64
5.12 Estimated coverage of A3 - Cross-site Scripting (XSS) ................ 65
5.13 Estimated coverage of A4 - Insecure Direct Object Reference ........ 65
5.14 Estimated coverage of A5 - Security Misconfiguration ................ 66
5.15 Estimated coverage of A6 - Sensitive Data Exposure .................. 67
5.16 Estimated coverage of A7 - Missing function level access control .. 68
5.17 Estimated coverage of A8 - Cross-site Request Forgery (CSRF) ...... 69
5.18 Estimated coverage of A9 - Using components with known vulnerabilities .. 70
5.19 Estimated coverage of A10 - Unvalidated Redirects and Forwards .... 71
5.20 Estimated scores of all risks in OWASP Top Ten ........................ 73
5.21 Continuing: Estimated scores of all risks in OWASP Top Ten .......... 74
Chapter 1

Introduction

This chapter introduces the thesis by explaining its motivation, purpose, method, scope and structure.

1.1 Motivation and Background

Web applications are one of the most widespread platforms for information and service delivery over the Internet today\[38\]. They are increasingly used for critical services and may store sensitive information, which is why they have become a popular and valuable target for attacks. According to Verizon’s annual report\[57\], web applications are at the top in both the number of breaches and amount of data being compromised. A broken web application could result in enormous information breaches, severe financial losses and both ethical as well as legal consequences. Software systems are usually exploited by using vulnerabilities that depend on design flaws or bugs in the implementation. As indicated in Figure 1.1 they are typically introduced in the coding phase but detected later during the testing phase. At the same time, several authoritative studies have shown that defects during requirement engineering cost 10 to 200 times more to correct once fielded than during development\[5][41]\.

Web applications are subject to continuous changes during both development and once deployed, which is why they are often developed using rapid and lightweight agile approaches that allow reworking of requirements. A popular goal within many agile software projects is to establish automated build and release processes, so-called continuous delivery\[56\]. This requires setting up a chain of tools, the main being a continuous integration tool that automates the process of building and testing code at certain events or at specific moments in time. A necessary step to achieve continuous delivery is to verify that changes to the code satisfy the software’s requirements. Praxis in many software projects today is to conduct verification in continuous integration using automated unit, integration and system testing for functional, non-security related, requirements and ignoring the build if testing fails\[48\]. Usage of continuous integration in this way helps to constantly assure that systems does not have integration problems and that they are in functional states.

A continuous integration environment can be extended with various security testing techniques in order to expand the testing scope to include security requirements. However, incorporation of software security in lightweight, agile, approaches can be difficult. To use
security testing throughout a complete agile development process by running automated tools in continuous integration environments is one approach that strives to improve security in agile projects. Instead of performing security testing at the end of development cycles, which often is the standard scenario, such methods enable early and continuous detection of security risks and vulnerabilities.

The purpose of this thesis is to study how existing security testing techniques operate in continuous integration environments and which security requirements that can be verified by using such techniques. The work is a qualitative analysis of different security testing techniques and aims to evaluate how they technically fit into a continuous integration environment as well as how they adhere to agile principles. The thesis is a collaboration between the author, Pontus Thulin, and the consultant company Omegapoint and aims to address integration of security testing techniques into continuous integration as a means towards automated security requirement verification in continuous integration.

1.2 Purpose

The purpose of this thesis is to investigate which, and to what extent, security requirements are possible to verify in continuous integration processes for web applications, using existing security testing techniques.
1.3 Framing the Problem

The nature of the thesis’ purpose first obliges research of several different fields within software development processes and software security. The main problem in this thesis is to investigate these areas and to determine a way of comparing the identified security testing techniques with each other. Which testing technique enables verification of which security requirements and how well do they fit in continuous integration environments?

In order to establish which kinds of security requirements that are possible to verify with existing security testing techniques, some instrument for comparison is required. For this purpose OWASP Top Ten (OWASP Top Ten is further discussed in section 2.5 at page 12) was identified as a tool that enabled comparison of which of the most important security risks that each testing technique can help mitigating.

Furthermore, studying how well and to which extent existing security testing techniques operate in continuous integration environments requires investigations of the field of agile software development processes, and more precisely continuous integration. By studying the nature of agile development and continuous integration, a set of desirable properties (these are discussed in section 3.7 at page 22) are elaborated. The purpose of these properties is to enable comparing how well each technique is suited for testing in such environments.

Finally, the key part of the thesis is to research the field of security testing techniques; which techniques exist and how to they function? Based on this information, the techniques can be compared and analyzed with the previously established instruments.

1.4 Scope

As discussed in section 1.3, this thesis concerns several areas within the software security field such as security requirements, security testing and agile security development methodologies. All of these fields consist of many different methods and theories. To research all of these is too exhaustive for this thesis, which limits the scope on which to lay focus. In some sense, not being able to conduct an exhaustive study will lead to an arbitrary selection of sources of information.

Applications' infrastructure gets increasingly complex and interconnected and the difficulty of achieving security in software increases exponentially[18]. Internet-enabled software applications are commonly exploited targets[42]. For this reason as well as the fact that web applications are often developed using agile methods, the scope of this thesis is limited to web applications. Some of the security testing techniques discussed however also applies for native applications without network connections. Although, such applications would also require additional testing techniques due to other kinds of problems being introduced when the client has access to the source code of a running system.

As many of the commonly used security testing tools on the market are commercial and expensive, the evaluation of security testing techniques will be qualitative. Analyzing open-source security testing tools does not provide a fair comparison of the techniques, as such tools often are inadequate in their findings compared to commercial market leaders. Furthermore, in order to conduct a fair comparison between techniques one should preferably incorporate several tools from each of the security testing techniques identified, which is a process too time-consuming for the scope of this thesis. The qualitative evaluation is
1.5 Method

To accomplish the purpose of this thesis, a qualitative method was used. The qualitative method is based on a theoretical understanding of the area with only a few practical examples. A qualitative analysis is contrary to a quantitative analysis, where the analysis is rather supported by practical, statistical and mathematical data. The conclusions of qualitative analysis methods are in general difficult to generalize to other domains but can introduce new hypothesis that can be verified using qualitative methods. The qualitative method was chosen to introduce the subject and to provide a wider scope than what’s possible with a quantitative analysis with the same time limit.

1.6 Outline

The outline of this report is based on the use of a qualitative method. The report is split into sections of theory, analysis and conclusion. The theory section investigates the three main areas; Security Risks and Requirements (chapter 2 at page 8), Security in Agile Software Development Processes (chapter 3 at page 16) and Security Testing Techniques (chapter 4 at page 26). A qualitative analysis is then performed from the gathered information, evaluating each identified security testing technique in order to establish how well they are suited for continuous integration environments. Following is a structured description of the outline.

1. (Theory, chapter 2) Gain knowledge about security risks and requirements, and decide how to evaluate and compare different security testing techniques by which requirements or risks it can verify.

2. (Theory, chapter 3) Gain knowledge about agile development and continuous integration. From this, establish properties that are required of security testing tools to be able to integrate into the testing and verification process of continuous integration.

3. (Theory, chapter 4) Study the nature of vulnerabilities and risks in web applications and gain knowledge about existing security testing techniques. Research how the identified security testing tools function, what kinds of vulnerabilities they can detect and establish a suitable categorization of the tools that fits the purpose of this thesis.

4. (Analysis, chapter 5) Qualitative analysis of the different security testing techniques that were identified in the litterature study. The first part analyze by how well they fit into continuous integration environments by comparing with established properties (analyzed in section 5.1 at page 49). Further, use the findings in chapter 2 to analyze which security risks that could be avoided or reduced with the various security testing techniques (analyzed in section 5.2 at page 61).

5. (Conclusion, chapter 6) Discuss the analysis based on the purpose. Is any security testing technique suitable for usage in continuous integration environments? What level of security is possible to assure?
Figure 1.2 further clarifies how the method is meant to support conclusions derived from the purpose.
1.7 Related Work

Security Testing of Web Based Applications\cite{14} is a master thesis written for NTNU by Gencer Erdogan. It states that web based applications are developed using agile approaches and contains a practical implementation of dynamic security testing in an agile project. However, the thesis neither discusses continuous integration nor compares how different security testing techniques fit into agile projects.

Building Security In Using Continuous Integration \cite{21} is a short report written by Thomas Stiehm and Gene Gotimer. It concludes that security can be built into agile projects using automated security testing. However, it neither attempt to compare how well different techniques fit into such environments nor determines which level of security it can assure.

When writing this thesis, no study that compares how different security testing techniques fit into continuous integration were identified. However, there exist several practical attempts to use distinct security testing techniques within a continuous integration environment. The following lists displays some examples of such.

- Intellavis.com published a comparison of various dynamic security testing tools, here referred to as dynamic application security testing (DAST)\cite{11}.
- OWASP Zed Attack Proxy, ZAP, published a video showing how to use ZAP to perform security regression tests\cite{62}.
- Jenkins is a well-known continuous integration tool that has plugins for various static application security testing (SAST) tools\cite{33}.

1.8 Evaluation of Sources

The references used for this thesis are from a wide variaty of sources. The chapter about security requirements (chapter 2) is mostly of a theoretical nature and thus founded on many academic papers, discussing various theories. As there are several possibly correct answers to how security requirements should be managed, many papers advocate different opinions. Academic papers however always establish well-documented resources and are for this thesis thought of as relatively trustworthy.

The chapter regarding software development and software security (chapter 3) discuss a field that depends on more facts and less opinions. There are however some different theories and methods regarding how distinct features of development processes should be handled but they are often based upon the same basic ideas. Many of the used sources are either well-known books or academic papers within the field.

The chapter about security testing techniques (chapter 4) deals with a domain that consists of many opinions about how security testing should be performed. Some books and academic papers were found concerning SAST and DAST approaches, but none were identified for interactive application security testing (IAST). The lack of academic papers within this field has led to the need of gathering information from blogs, vendors of tools, and lectures. This fact has to be taken into account when reading or using information from this thesis.
1.9 Acknowledgements

Foremost, I would like to articulate my genuine gratefulness to my advisor Sebastian Åkerman at Omegapoint. This thesis is based on his observation of a lack of verification of non-functional security requirements in agile development and especially continuous integration environments. He has contributed with both a vast amount of knowledge in all fields that are regarded in this thesis, as well as with his time with reading and correcting the report.

Besides Sebastian, I would also like to thank the rest of Omegapoint. I’ve been received and welcomed with kindness, and many individuals have contributed with their time and knowledge answering my questions.

I would also like to thank Ulf Kargén and prof. Nahid Shahmehri at Linkopings University. Ulf has been both very patient and skillful in guiding me with the writing of this report. Nahid, with her profound skills in computer security, has provided sharp and proper critique in order to guide me in the right direction.
Chapter 2

Security Risks and Requirements

It seems to be an agreed fact amongst application security experts that security requirements are important to both define and manage in order to enable effective evaluation of the level of security achieved in software\cite{28}\cite{42}. However, there is a lack of agreed definitions, and the methodologies for dealing with security requirements varies\cite{34}.

This section addresses how security requirements differ from non-security related software requirements by often being non-functional. Non-functional requirements are difficult to verify using functional testing methods such as unit\footnote{Unit testing tests individual components of a system. It is usually conducted by sending input data to functions and asserting that the result is as expected}, integration\footnote{Integration testing tests the integration between software components by combining them to groups and running tests} or system testing\footnote{System testing is conducted towards the end of development processes and aims towards testing a complete system} and thus require alternative approaches. Often, non-functional security requirements are managed using a risk-based approach, which is also addressed.

Software requirements are usually split into functional and non-functional requirements, which can also apply in the security field. Functional requirements describe functionality of systems; what a system should do, while non-functional requirements treat questions regarding how functionality should behave. In the literature, security requirements are sometimes only referred to as non-functional, but they can relate to both.

2.1 Functional Security Requirements

As mentioned, functional requirements in general describe functions that specify what a system should do. Functional security requirements are of the same kind, defining functional aspects of software security and are often related to usage of security controls. A security control is a countermeasure implemented to mitigate security risks, such as implementing SSL\footnote{Secure Socket Layer (SSL) is a cryptographic protocol used to provide secure communication over the Internet} to protect against eavesdropping and data manipulation. The functional security
2.2 Non-Functional Security Requirements

The term non-functional requirement, NFR, has been in use for more than 20 years, but no agreed definition within the requirement engineering community exist\[20\]. The lack of a definition is related to the fact that non-functional requirements often are being thought of as fuzzy and abstract and are therefore difficult to model, verify and test. This has led to non-functional security requirements regularly being neglected\[10\], but at the same time there seems to be a unanimous consensus that non-functional requirements can be critical for the success when developing software\[20]\[10\].

Non-functional security requirements are a subset of non-functional requirements, where the main dissimilarity lies in the context of why the requirement is needed. Both describe how a system should behave in certain situations, with the difference that security related non-functional requirements involve some kind of threat that should be avoided. Unlike functional security requirements, non-functional security requirements are not strictly related to security controls. Regardless if it concerns the inside of a security control or some non-security related functionality, a non-functional security requirement can be defined to assure some level of security within the section of the software that the requirement concerns. McGraw explains this with the term *software security is not security software*, meaning that a security flaw can be critical regardless of whether it exists within a security feature or in the noncritical GUI\[42\]. Security features, defined by functional security requirements, are obviously needed when securing a system, but as well as any other feature of a system it should be defined how they should work to assure security.

2.3 Requirement verification

Verification of software requirements is done by analyzing requirements on their own by conducting different kinds of tests, assuring that the software behaves as defined. Tests could be split into two domains, depending on whether it’s a functional or non-functional requirement that is to be verified.

Verifying a functional security requirement involves ensuring that the security control is actually implemented and put in place, which can be done by unit or integration tests. This functional type of testing is often referred to as positive testing and aims to verify that valid inputs result in expected outputs, but also that a system graciously handles invalid inputs by for example displaying proper error messages\[7\]. Once functional tests are implemented they can easily run again and used as regression testing in continuous integration environments. As functional security requirements possibly are verified using the same testing techniques as for any type of functional requirement, it will not be further discussed in this thesis.

Verification of non-functional security requirements is a more difficult and arbitrary process, as there is no expected behavior to look for. As non-functional security requirements relate to how software should behave when not used as intended, verification requires security testing to shift the tester’s mind-set from verifier to attacker\[7\].
Figure 2.1: Verification of security requirements

Figure 2.1 attempts to visualize the difference between assuring functional and non-functional security requirements. Functional testing is represented by the green arrows, which are tests aiming on assuring that a certain component is actually put in the correct place and that it is up and running when used as intended. The red arrows are tests that have the purpose of assuring non-functional requirements by running tests outside of the normal scope. Acquiring full testing coverage is generally not possible, as it would require testing all combinations and variations, which is an exhaustive task for any moderately complex program. The art of software security testing[7] explains this with the following example.

Imagine a Web application that is made up of 10 forms with 10 form fields each. Each of the fields can take an input of 100 alphanumeric characters. The total number of input possibilities for this program is \(10 \times 10 \times 62^{100}\) possible inputs.

Providing proof of complete security would require testing of all potential attacks and all execution possibilities. Obviously that is an unmanageable task, which is why security testing often is based on a risk engineering approach. The risk-based approach allows testing to test the interfaces and functionality that an attacker is most likely to go after first[42][7]. Several standards for managing security risks have been published, explaining how risks should be managed[49].

2.4 Security Risks

OWASP Top Ten[18] defines security risks as vulnerable paths through an application that attackers can use to do harm to the business or organization. Each of these paths represents a risk that may or may not be serious enough to warrant attention. This is illustrated in Figure 2.2.

The existence of unwanted paths through an application is caused by vulnerabilities. CISSP[22] defines vulnerability as follows,
2.4. SECURITY RISKS

“A software, hardware, procedural or human weakness that provide an attacker the open door he is looking for to enter a computer or network and have unauthorized access to resources within the environment. The vulnerability characterizes the absence or weakness of a security control that could be exploited. This vulnerability may be a service running on a server, an open port on a firewall…”

A technical definition of vulnerabilities and how they can get introduced into software is discussed in section 4.1 at page 26.

The seriousness of risks depends on the potential impact and probability. As illustrated in Figure 2.3, the risk is the intersection where an attacker or some other threat is taking advantage of a vulnerability and the corresponding business impact on an asset if successful. Thus, threats may exist, but if there are no vulnerabilities to take advantage of the risk it would not be exploitable. Similarly, software can have vulnerabilities but no threats or assets without having a risk.

In order to mitigate potential risks, security controls are often put in place. Security controls can take the form of a software configuration, a hardware device or a procedure and has the goal of reducing or eliminating the likelihood or consequence of vulnerabilities being exploited. Examples of security controls can be software implementations such as input validators, cryptographic algorithms or authentication mechanisms. It should be noted that security controls, as well as any other software component, could itself contain vulnerabilities that can be exploited by attackers. Establishing security controls is the first step towards protection, but as addressed in section 2.3, the quality of the controls also has to be assured by defining non-functional requirements. There are several examples of security controls where the quality was not assured, leading to devastating consequences. In 2014, both the Heartbleed and Goto Fail bugs are practical evidence of such.

When discussing security risks and vulnerabilities, OWASP Top Ten and SANS Top 25 most dangerous software errors are commonly used as starting points. They are lists that gather the, according to them, most commonly existing and exploited vulnerabilities in software.

Figure 2.2: Risks as paths through a system[18]
For web applications, OWASP Top Ten is one of the most commonly used, and many standards, books, tools and organizations refer to it\cite{mitre}. As the scope of this thesis only concerns web applications, OWASP Top Ten will be chosen and used as an instrument to enable comparison between different types of security testing techniques and which vulnerabilities they can detect. In the analysis section, each identified security testing technique is evaluated by which of the vulnerabilities in OWASP Top Ten that it can possibly detect. The list is not comprehensive, but is usually thought of as a good starting point when securing a web application. Following is an explanation of OWASP Top Ten.

2.5 OWASP Top Ten

OWASP Top Ten\cite{owasp} is a well-recognized and referenced list of critical security risks. The list is deduced from data spanning over 500,000 vulnerabilities across hundreds of organizations and thousands of applications. Items in the list are prioritized according to occurrence in combination with estimates of exploitability, detectability and impacts.

There exists lists similar to OWASP that could have been used for the purpose of comparing security testing techniques. SANS Top 25 is a well known list that maps directly to CWE weaknesses, making it more concrete than OWASP Top Ten. Further, OWASP Top Ten describes risks at a more general level, which is why most of the weaknesses described in SANS Top 25 maps well into a risk in OWASP Top ten. For this thesis, OWASP Top Ten

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{security-risk.png}
\caption{Security risk}
\end{figure}

\begin{itemize}
\item[\textsuperscript{7}] MITRE, PCI DSS, DISA, FTC, and many more.
\item[\textsuperscript{8}] Common Weakness Enumeration (CWE) is a community project that provides a unified, measurable set of weaknesses in software.
\end{itemize}
was considered as better suited than SANS Top 25, or some other less known collections. The fact that OWASP Top Ten describes risks at a high level works well with the high-level descriptions of security testing techniques used later in the report. Also, analyzing twenty five risks instead of ten would make the analysis section unnecessarily exhaustive.

Following is a short description of each entity in OWASP Top Ten; how each risk can be exploited as well as how they can be detected.

2.5.1 A1 - Injection
Injection vulnerabilities are common in conjunction with many interpreted languages\(^9\) used in web applications. The injection occurs when untrusted data, typically user supplied, is sent to an interpreter and used as part of an executed command. The user-supplied data can fool interpreters into executing chosen commands.

Injection vulnerabilities are usually detected by either finding vulnerable paths in applications where user supplied data is used by interpreters or actually trying to inject data and analyze responses.

2.5.2 A2 - Broken Authentication and Session Management
Weaknesses in implementation of authentication can enable attackers to compromise and access information that belong to other users. Vulnerabilities are often related to weak management of session tokens or credentials. OWASP lists the following causes of vulnerabilities.

- User authentication credentials aren’t protected when stored using hashing or encryption. See A6.
- Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs).
- Session IDs are exposed in the URL (e.g., URL rewriting).
- Session IDs are vulnerable to session fixation attacks\(^{10}\).
- Session IDs don’t timeout, or user sessions or authentication tokens, particularly single sign-on (SSO) tokens, aren’t properly invalidated during logout.
- Session IDs aren’t rotated after successful login.
- Passwords, session IDs, and other credentials are sent over unencrypted connections. See A6.

Some session-related issues can be detected by analyzing session handling in network communication, while others such as not using safe storage of credentials require inside knowledge of applications.

---

\(^9\)Interpreted languages, such as JavaScript and SQL, execute instructions directly without previous compilation to machine code.

\(^{10}\)Session fixation is a weakness where attackers are allowed to fixate other users’ session identifiers, making it possible to gain control over their sessions.
2.5.3 A3 - Cross-site Scripting (XSS)

XSS is a specific type of injection and therefore has similarities with A1. A XSS attack allows untrusted data to be sent to browsers without proper validation. There are four different kinds of XSS: Stored, Reflected, DOM-based and Mutated. The first three occur at server-side while mutated XSS (mXSS) is a relatively new kind of attack that bypasses server validation using automatic correction functions in clients’ browsers. XSS allows attackers to inject client scripts into web pages viewed by users and thus execute malicious code. Many XSS vulnerabilities are possible to detect automatically in the same manner as A1. However, each application is built differently and may use interpreters such as JavaScript, ActiveX, Flash and Silverlight, which makes complete coverage with automated testing difficult.

2.5.4 A4 - Insecure Direct Object Reference

Insecure direct object reference, IDOR, vulnerabilities occur when an internal object, such as a file or database object, gets exposed without proper access control checks. If users are able to access certain objects without proper authorization checks, it can be possible for attackers to get references to these objects by manipulation of input parameters such as a filepath, key, URL or form parameters.

To test whether an IDOR attack is possible requires knowledge about which users that are allowed to access which objects within a system. Such information is often very specific to each application and is therefore generally difficult automating.

2.5.5 A5 - Security Misconfiguration

Misconfigurations can happen at any level of a system. It is a wide term that includes outdated software usage, unnecessary enabled features, default accounts, overly informative error messages and unsecure settings in development frameworks.

Some security misconfigurations can be detected from the outside by using public information, such as usage of weak SSL or web-server related issues. However, detection of most vulnerabilities requires access to internal configuration files.

2.5.6 A6 - Sensitive Data Exposure

Sensitive data exposure vulnerabilities are related to certain types of data that for some reason requires extra caution. This can, for example, be passwords, credit card numbers, health records, or personal information. A6 vulnerabilities usually depend on missing or weak usage of encryption when sending and storing data.

Detecting A6 vulnerabilities requires knowledge about which data that is sensitive, making it difficult automating. However, some vulnerabilities are possible to detect by tests from the outside, such as sending unencrypted login request as such signatures are often generic in their appearances. It is also possible to detect some vulnerabilities from the inside, by finding usage of standard APIs in unsecure manners. For example, it’s possible to know of any hash API is used when storing something named ‘password’.
2.5. **A7 - Missing function level access control**

A7 has some resemblance to A4, but deals with function access instead of objects. Missing function level access control enables attackers to access functions within a system, to which they are unauthorized. This can for example be performed by changing a function parameter that is sent to the server, executing optional methods.

As for A4, detecting missing function level access control vulnerabilities requires knowledge about which users that are privileged to do what and is therefore difficult automating.

2.5.8 **A8 - Cross-site Request Forgery (CSRF)**

CSRF, also known as one-click attacks or session riding, are attacks where unauthorised requests from one site are transmitted to another, already authenticated, site in the same browser. The web application receiving the forged requests believes that it is valid, as the user is already authenticated and has an open session.

One simple way of testing whether an application is vulnerable is to see if any links and forms lack an unpredictable CSRF-token or other types or re-authentication such as CAPTCHA. Such methods however can result in a high amount of false positives (explained in section 4.3.4 at page 36) as it might report missing CSRF protection for requests where it’s not required.

2.5.9 **A9 - Using components with known vulnerabilities**

As most applications developed today depend heavily on third party components, it’s important to make sure that the external components don’t include vulnerabilities. As the heading indicates, A9 only concern those components that are already known to be vulnerable.

Testing for A9 requires a black list composed of which components that are known to be vulnerable and then comparing with the ones used within the project. As long as testing tools have access to the list of used components, automating A9 detections is straightforward. However, the black list of known vulnerable components has to be continuously up to date to be useful.

2.5.10 **A10 - Unvalidated Redirects and Forwards**

Applications frequently redirect users to other pages. If the target site for a redirect is specified in an unvalidated parameter, it allows attackers to choose malicious destinations by providing URLs of their choice. The best way to avoid A10 is to not use redirects or forwards at all. If redirects has to be used, it should not include parameters supplied by users when calculating the destination or at least have them properly validated.

Detection of A10 can be done in multiple ways. With access to source code, one can look for usage of calls to redirect or forward APIs and make sure that no user parameters are included in the URL. If such parameters are used they should be validated through a whitelist to not be vulnerable. Another approach can be to let spiders crawl the application to see if it generates any redirects. If the parameters of the request prior to the redirect appear to be a target URL or a piece of it, the application might be vulnerable.
Chapter 3

Security in Agile Software Development Processes

Continuous integration is an agile technique; implemented to achieve agile practices defined in the agile manifesto. Integrating security into continuous integration thus requires an understanding of both agile principles as well as general security development methodologies, which is addressed in this chapter.

3.1 Software Development Processes

A software development process, sometimes referred to as Software Development Life-cycle (SDLC), is a method aimed towards imposing structure into the development of software products. Existing SDLC methodologies are usually split into two opposite camps; the sequential Waterfall\(^1\) model and the iterative agile approach. Within these there exist several different methodologies, each describing more explicitly how development should be approached. This thesis however only focus on the high level properties of the waterfall model and especially agile development. As illustrated in figure 3.1, agile software development consists of many iterations making it adaptive to changes. The agile approach is based upon agile principles, stated in the agile manifesto.

\(^1\)Waterfall is a sequential development process, in which the progress in seen as flowing downwards through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Implementation and Maintenance. When working with the waterfall model, changes in finished phases should be avoided.
3.2 The Agile Manifesto

The agile principles were stated in The Agile Manifesto[3] in February 2001. There are twelve principles, stated as follows;

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

7. Working software is the primary measure of progress. Agile processes promote sustainable development.

8. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity—the art of maximizing the amount of work not done—is essential.

11. The best architectures, requirements, and designs emerge from self-organizing
teams.

12. At regular intervals, the team reflects on how to become more effective, then
tunes and adjusts its behavior accordingly.

3.3 Software Security

Software security is a relatively young sub-domain within software development, that deals
with the process of engineering software that continues to function correctly under malicious
attacks. There are several steps that needs attending when working with software security,
one of the most important is introducing security phases into the software development
process, a so called secure development life cycle.

3.4 Secure Development Life cycle

Many within the software security field argue that security has to be built in throughout
the whole development of software. This has led to evolvement of several methodologies
that integrates security into software development processes, so called Secure Development
Life-cycles, SDLCs. Microsoft’s SDL[25], Secure Software Development Lifecycle (SSDL)[7]
and Cigital’s Touch Points[42] are usually mentioned as the most common. All of these
practices build on integrating security into already established software development lifecy-
cles by applying security practices into the different phases of development. The three are
of sequential structures, mapping neatly to the phases of the Waterfall model[44].

3.5 Software Security in Agile

One of the most obvious contradictions between agile development methodologies and soft-
ware security is that software security often is thought of as difficult, time-consuming and
expensive, while agile embrace the opposite. Many argue, including security people them-
selves that security should be done by security experts. Agile on the other hand works
towards self-organizing teams where simplicity is advocated and expertise is kept within the
team where potential problems preferably are solved internally.

Testing and verification of security requirements is one of the areas where software security
could be argued being most difficult and time-consuming. Usually, security testing is con-
ducted at the end of the project, often by hiring external penetration testers who detect
vulnerabilities in a system without correlation to an application’s specified requirements.
One way of integrating security testing throughout a complete, agile, development process
can be through automation using tools[21]. Tools have the power of both including se-
curity knowledge and speeding up processes, allowing a shift towards security testing and
integration being done by developers throughout the whole development process, instead
of ad-hoc security teams. Continuous integration pushes towards testing with automated
tools, making it a good place to also integrate testing and assurance of security[24].
3.6 Continuous Integration

Achieving continuous delivery of quality code that works and is reliable depends on implementing continuous integration, CI[12], which is a system for integrating and testing code continuously. Continuous Integration is an agile technique that is used in 56% of agile projects according to a survey conducted by VersionOne[56]. CI is a software development practice that requires team members to integrate their work frequently, continuously ensuring that software components work together. Each developer integrates code at least daily, leading to multiple integrations each day. Integrations are automatically built by a build server and verified by running regression tests to detect errors as quickly as possible[48]. The goal of CI is to reduce integration problems, which are common in large projects with multiple developers working on shared code, and to enable developing of cohesive software more rapidly. CI fits well into several of the principles in The Agile Manifesto, such as

- Customer satisfaction by rapid delivery of useful software
- Working software is delivered frequently
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily cooperation between business people and developers (developers being able to show working software helps cooperation between development and operation)

![Figure 3.2: Illustration of a Continuous Integration environment](image-url)

Figure 3.2 illustrates the process and implementation of a CI system. The CI server has the possibility to run PRE and POST build scripts, allowing execution of eligible external tools.
Further, the final goal of continuous integration is often to reach the state of continuous delivery. Continuous delivery is a practice or philosophy that work towards a state where it is automatically assured that the code always is ready for release[54]. The first three levels in Figure 3.3 (automated fetching, building and unit testing) are managed by implementing a continuous integration process, but to realize continuous delivery, acceptance testing of non-functional requirements is required. Such testing, amongst others, aims towards ensuring both quality and security of systems. Usually, security testing is conducted as a manual step, prohibiting a full chain of continuous delivery. Including security testing into the continuous integration system is a step closer towards a fully automated continuous delivery chain.

**Figure 3.3: Illustration of a Continuous Delivery chain[6]**

### 3.6.1 Security Testing in Continuous Integration

Today, the standard way of performing testing in a continuous integration process is to run functional testing in the form of unit, integration and system tests, verifying functional requirements. Extending testing to also assure security in continuous integration environments requires risk-based security testing, which obliges running automated tools designed for this purpose. The techniques used by such tools are further discussed in chapter 4 at page 26. Continuously testing for security vulnerabilities in a continuous integration environment could help ensuring that security is built in throughout the whole development process by verifying that new or changed functionality does not add new weaknesses.

Figure 3.4 illustrates how security testing tools can be run in the testing environment in automated builds. The build scripts sets up a running version of a system within a testing environment, to which external testing tools can be added. Figure 3.5 shows a sequential
3.6. CONTINUOUS INTEGRATION

Figure 3.4: Illustration of security testing in CI

process of security testing tools in continuous integration.

Figure 3.5: Sequential illustration of security testing in CI[51]

However, integrating external security testing tools into continuous integration environment is not always straightforward. The tools have to both fit technically as well as adhere to agile principles in order to be appropriate for such usage. In order to evaluate how well different security testing techniques are fit for incorporation in continuous integration, a set
3.7 Wanted properties in Continuous Integration

The list of properties that follow is non-exhaustive and derived from general technical requirements of continuous integration systems as well as agile and continuous delivery principles[26]. Its purpose is to be used as an instrument when analyzing, evaluating and comparing how well the identified security testing techniques are suited for continuous integration environments.

3.7.1 Automated execution

3.7.1.1 Rationale
Builds should be fully automated[48] and run as frequent as possible. This requires the whole process to be automated, including execution of external tools.

3.7.1.2 Criteria
The possibility of automated execution is determined by deducing if any manual steps are required to run the testing.

3.7.2 Headless

3.7.2.1 Rationale
A CI server needs a headless process, such as a single command script, to execute in an automated manner[48]. This means that the whole process should be able to run without any graphical interfaces, including external tools.

3.7.2.2 Criteria
Deduce if it is possible to execute external tools from command line or a CI plugin\(^2\). All desired commands and responses should be available this way for the headless property to be applicable.

3.7.3 Repeatable

3.7.3.1 Rationale
One of the required qualities of continuous testing in CI is to make tests repeatable[48][26]. This property is required for security testing to become a valid regression test.

Suppose that a build fails due to some test that did not pass. If the functionality regarding this test is fixed, the test needs to run in exactly the same way to provide information that the problem is actually resolved. If the test is not repeatable and runs in a different manner,

---

\(^2\)A CI plugin allows external tools to be easily integrated and executed automatically within the CI environment.
one does not know if the functionality was fixed or if the test just behaved differently and consequently failed detecting the problem.

3.7.3.2 Criteria

External security testing tools have to run deterministically. This means that execution of tools should not include required randomness, and if so, the random seed must be possible to set to the same value between different builds.

Further, tools that make use of databases when testing systems has to make sure that databases are initialized with the same data between different test runs. There exist several approaches for managing test data generation\[37\], and as long as they’re deterministic, it’s outside of this thesis’ scope which to use.

3.7.4 Internal expertise

3.7.4.1 Rationale

The agile manifesto advocate self-organizing teams with motivated and trusted individuals as well as simplicity[3]. Thus, when running security testing tools daily in automated builds the results has to be interpretable by developers without deep security knowledge, so that no external expertise is required for daily tasks and decisions.

3.7.4.2 Criteria

A relevant factor to consider can be if the tool provides distinct information about how the vulnerability can be exploited, where it exists in code and how it can be fixed. Such information facilitates in resolving security issues for developers without knowledge about software security.

To check whether reported issues correspond to real threats or not can be difficult, which is why false positives may require more security knowledge of developers. The amount of false positives should therefore be kept to a minimum to reduce the need for external security expertise.

All existing security testing techniques require fine-tuning by defining rules for what to check for and also which findings to exclude. This kind of work should preferably be kept to a minimum in order to be easily managed by developers.

3.7.5 Automated decisions

3.7.5.1 Rationale

As discussed in section 3.6 at page 19, the final goal of CI often is to reach the state of continuous delivery. To achieve continuous delivery, security testing has to result in information that enables decisions. Automated decision making is required to accomplish the continuous delivery principle of automate everything[26], which means that the verdict of deciding if the build should pass or not cannot require manual steps. However, the results from security tests correlate to non-functional requirements and are therefore not binary. This means that the results are sets of information that reflects the security state of a system, rather than a passed or not passed as when running functional tests. To interpret
such results, the data has to be processed and adjusted for the specific system using security metrics. This procedure is further discussed in section 4.1.3 at page 29.

Regardless of which security testing techniques that is used, it has to fulfill a set of criteria that makes the data reliable. If the results from testing cannot be trusted, automated decisions won’t be possible.

3.7.5.2 Criteria

This property mainly relates to how results are presented, the level of trustworthiness of the results, and to which extent testing was conducted.

Preferably, the security testing techniques present results by different level of severity and reliability of found vulnerabilities. Such levels can be used when defining specific levels of acceptance of security issues within a project, such as defining rules saying that no medium or high vulnerabilities are allowed.

Further, to automatically conclude whether a build is secure or not, more or less the whole system should be tested. This requires the testing to obtain a high test coverage as well as providing a wide range of possible findings. In order to make automated decisions, there is also a need of getting information about which sections of a system that were tested and which were not.

It should also be established that the results from a security testing technique has a high level of trustworthiness. When discussing security testing, trustworthiness implies that all, or at least most, of the vulnerabilities that a certain technique claims to detect are found. Furthermore, it also means that the found vulnerabilities are reliable in the sense of not being false positives. This is a difficult composite achieving and is further discussed in section 4.3.4 at page 36.

3.7.6 Fast execution

3.7.6.1 Rationale

A healthy CI process requires rapid feedback from builds. Tools that run quickly will therefore make a better fit in continuous integration systems.

Continuous integration tools will enable different builds depending on the event. Security tools with fast execution time can be run more often, for example with each commit, while other that require longer execution times can run nightly or at weekends.

It is worth noting that the testing time of external security testing tools generally depend on system size. As applications grow in size, execution time increases and the accepted time of testing has to be relative to this.

---

3test coverage, or code coverage, refers to a measure used to describe the degree to which a system is tested. A high test coverage means that the majority of the system’s code was executed and tested, which is desirable for the property of automated decisions. It should be noted that there are several different variations of the measure and which one that should be used is outside of this thesis’ scope.

4Possible findings refers to which vulnerabilities that the different tools enables finding.
3.7. WANTED PROPERTIES IN CONTINUOUS INTEGRATION

3.7.6.2 Criteria

Confirm how time-consuming execution of the security testing technique is. The criterion is then simple; external tools should execute as rapid as possible to fulfil this property.

3.7.7 Early testing

3.7.7.1 Rationale

As discussed in section 3.5 at page 18, the goal of security testing within continuous integration is to support building security into software throughout the whole SDLC[42]. If testing cannot be included until the end of development, it will not add any benefit of being run continuously during development and could as well be thought of as an ad-hoc penetration test.

Further, testing should be continuously conducted throughout the whole development process.

3.7.7.2 Criteria

Confirm when and how in the SDLC the security testing techniques makes sense using. The earlier in the development process, the better fit for continuous integration. Testing should also be done as often as possible.

3.7.8 Simple integration

3.7.8.1 Rationale

Agile pushes towards simplicity and frequent delivery of working software[3], which is why external tools should be easy and fast to integrate into CI systems.

3.7.8.2 Criteria

Installing and fine-tuning tools into the CI environment will always require manual work. These tasks cannot be completely avoided, but should be kept to a minimum to fit into the CI process. How difficult a certain technique is to integrate can be deduced from how it functions. Does it require a running test environment? Does it require including mock data in the database? Does it require continuous fine-tuning after installation? Is the command line interface or API easy to understand and use?
Chapter 4

Security Testing Techniques

Security testing aims towards finding unwanted behavior outside the scope of functional requirements, which obliges additional testing techniques besides unit, integration and system testing. As discussed in section 2.3 at page 9, managing behavior outside of the functional scope requires non-functional security requirements and risk-based testing. All security testing techniques explained in this thesis use different kinds of risk-based testing approaches by focusing its testing on known, common, and critical kinds of vulnerabilities in software.

4.1 Vulnerabilities, Bugs and Flaws

Software vulnerabilities are unwanted system behavior that attackers can exploit and fall into one of two major categories – bugs at the implementation level or flaws at the design level\[42\]. From the attacker’s perspective, he or she generally does not care whether the vulnerability is due to a flaw or a bug, although bugs tend to be easier exploiting\[43\].

A flaw is a mistake in the design that prevents the program from operating securely, no matter how perfectly it was implemented. Design vulnerabilities are usually found within the software’s security features, such as encryption keys not being securely exchanged or based on improper secrets. It is worth noting that security flaws can exist in any component of a system or depend on complete absence of security features, such as lack of validation at user input sources.

Incorrect coding of software causes implementation bugs. Typically they are errors such as not checking return codes, not sizing buffers properly, not handling unexpected input properly. Usually, the bugs are really tiny, such as incorrectly calculating a size that is off by 1 or being off by a factor 2 because someone forgot that a string variable was in Unicode characters, which uses 2 bytes per character\[7\]. Even though bugs might be due to small mistakes, the consequences can be catastrophic.

Vulnerabilities are seldom due to faults in single components, but rather a series of events that together forms vulnerable paths that may result in exploitability (Illustrated in Figure 4.1). For example, being able to insert JavaScript\(^1\) into a database is not really a vulnerability, unless it’s later possible to present it to other users and thus execute the script.

---

\(^1\)Javascript is a dynamic programming language, most commonly used by web browsers at the client side.
Detecting such vulnerable paths can difficult and requires knowledge about how components of systems interact as well as which paths that can be dangerous.

The art of software security testing[7] presents the following lists of common design and implementation vulnerabilities. Section 4.1.1 and 4.1.2 presents a relevant subset of this list, aiming to clarify differences between design flaws and implementation bugs.

### 4.1.1 Common Design Issues

- Poor use of cryptography
  - Creating your own cryptography
  - Choosing the wrong cryptography
- Hard coded secrets
- Mishandling private information
- Tracking users and their permissions
  - Weak or missing session management
  - Weak or missing authentication
  - Weak or missing authorization
- Flawed input validation
  - Not performing validation in a secure context (such as at the client instead of the server)
  - Not centralizing validation routines
  - Not securing component boundaries
4.1.2 Common Implementation Issues

- Interpreted languages (Shell scripting/PHP)
  - Metacharacters
  - Command injection
  - Automatically created variables
  - Remote files
- Virtual machines (Java/C#)
  - Lack of error handling
- Platform Implementation issues
  - Symbolic linking
  - Directory traversal
  - Character conversion
- Generic application security implementation issues
  - SQL injections
  - Cross site scripting

As flaws are introduced during the architectural phase of software development, they tend to be deeper and more interconnected than implementation bugs. It could be argued that flaws in general can be more time-consuming to resolve than implementation bugs and therefore important to find early. Flaws do not depend on which programming language an application was implemented with, but rather relates to vulnerabilities concerning logical problems, privileges, weak design or mismanaged sessions. Compared to bugs, flaws are vulnerabilities that don’t necessarily result in distinct errors or crashes, but rather behavioral that is unwanted for some specific case. Determination of whether a system behavior is a flaw or a feature can be difficult to fully automate with tools, because one would require insight into privileges and intended behavior. However, some flaws are always unwanted behavior from a security perspective, independent of the situation, and can therefore be detected automatically. For example, if an application is designed to trust input from users without proper validation.

As seen in the list of common implementation issues, bugs can be language specific, platform dependent or general. Compiled languages such as C or C++ are the ones most haunted by bugs, as programmers handle memory management at a low level of abstraction. This allows the programmer to directly access memory, and if not done properly that can lead to a variety of serious vulnerabilities. In languages running in virtual machines, such as Java or C#, low-level memory is managed by the virtual machine. These languages are therefore mainly concerned with the general vulnerabilities seen in the list. These are often located within security mechanisms, such as bad implementation of input validation or session management, but can also exist in other non-security related components. Bugs are usually easier to find, as they often times result in distinct exploits such as modifying data in the database or shutting down a system.
4.1.3 Security Metrics

Security metrics is not a testing technique on its own and does not fit into the group of application security testing (AST) techniques described in the next section. It's rather a combination of measurements that together provide some business meaning. Measures are numeric values assigned to a given artifact or process. Software metrics in general are used to measure the quality of different properties of the software, thus often correlating to non-functional requirements. Security metrics are metrics that are coupled to aspects that in one way or another concerns security. Measures on their own do not directly provide relevant information, because all software projects are different and the information is relative. For example, “lines of code” or "Number of Vulnerabilities found" does not offer any individual information if not put into perspective. When combining the measures into a metric such as "Number of Vulnerabilities"/"Line of Code" a more interesting concept arise. Which security metrics that are possible to apply depend on the testing techniques used and what information that is gathered from these.

As security testing techniques correlate to non-functional requirements, the results from running tests are not binary but rather a set of information that reflects a system’s state regarding identified risks and vulnerabilities. From this information, a CI system can be able to deduce explicit decisions about the software’s security state. To enable making binary decisions from the results of security testing tools, there is a need of putting the results in perspective by adjusting results to each specific system. For example, if the result indicates a certain number of specific threats, it might be critical for one kind of system and not for another. This is where security metrics play an important role; by constructing measurements that in some way reflects the security state of a system. Using these metrics, each project can implement their own security rules that can be used for automated decisions. Figure 4.2 illustrates the process using metrics for automated decisions.
4.2 Application Security Testing Categories

Application Security Testing (AST) aims towards finding unwanted and vulnerable behavior outside of systems’ functional scope. Historically, the market of Application Security Testing evolved as two completely separate domains, Static (SAST) and Dynamic (DAST), which were used independently of each other. However, testing of next-generation modern web applications required a combination of the two, which now has begun reaching the market[40]. The combination of SAST and DAST is called Interactive Application Security Testing (IAST). The terms used in this thesis for different techniques might differ throughout the literature.

SAST, DAST and IAST describes security testing at a high level and differentiate depending...
4.3 Static Application Security Testing

SAST, or static analysis, tests applications for vulnerabilities by first constructing simplified models over possible executions paths and data flows, and then applying various techniques to analyse these models. The word static in SAST implies that analysis is done statically, i.e. without actually executing the programs. SAST is an inside-out, white-box, approach to

on the set of information available when testing. DAST is a kind of black-box testing method, that examines applications as users see them from the outside without knowledge about internal structures. SAST is the opposite, a white-box method that only test internal structures, and IAST is a grey-box approach that uses a mixture of both internal and external information. SAST, DAST and IAST are here split into sub-categories, explaining different technical approaches towards how vulnerabilities are detected. The list below of security testing methods only consists of techniques within the scope of the thesis; those testing web-applications that are currently under development. The list is non-exhaustive and derived from a variety of sources[19][36][55][9]. All identified techniques are put into one of the identified high-level categories, depending on the set of information accessible for each distinct testing-method.

The terms active and passive denote whether testing methods actively and consciously execute systems or not. Passive testing cannot test a system on its own, but requires active and external arrangements to execute it.

- **SAST** - White-box testing with information of applications’ internal structures
  - Source Code analysis
    Actively finds vulnerabilities by analyzing source code
  - Binary Analysis
    Actively finds vulnerabilities by analyzing binaries
  - Bytecode Analysis
    Actively finds vulnerabilities by analyzing bytecode

- **DAST** - Black-box testing with information of applications’ external structures
  - Penetration Testing
    Actively and passively finds vulnerabilities by scanning and attacking an application

- **IAST** - Grey-box testing with information of both internal and external structures of applications
  - Runtime analysis
    Passively finds vulnerabilities by analyzing a system from the inside, while it’s being executed
  - Combination of DAST and Runtime analysis
    Combines runtime analysis with DAST to detect vulnerabilities from both the outside as well as with knowledge of systems insides
  - Combination of SAST and DAST
    Actively finds vulnerabilities by mixing SAST and DAST techniques
4.3. STATIC APPLICATION SECURITY TESTING

application security testing that has access to the sources of applications, such as sources code, byte code or binaries. With access to these sources, SAST techniques foremost have information about applications’ internal structures, but can also obtain some knowledge about the external interfaces of applications. The information about external appearances however, does not include information only available in runtime, such as actual user data entering through these external interfaces, but rather about where different types of data can enter from outer sources.

SAST consists of three distinct testing techniques; Source code analysis, bytecode analysis and binary analysis. The differences between these lie in the sources used to perform analysis, but shares the methods applied to detect vulnerabilities. The shared methods and general concepts are explained to begin with, and the different testing techniques at the end.

4.3.1 Testing Scope

The main distinction between source code, bytecode and binary analysis lies in the scope of what can be tested as they use different sources, however, the categories of possible findings remain the same. The reachable scope depend on which of the analysis methods that is used, but one common property for all of them is that custom developed\(^2\) code and files are accessible. The different scopes are illustrated in Figure 4.3.

Following are the three distinct testing techniques further explained.

4.3.1.1 Source Code analysis

Source code analysis is the first and most common of SAST techniques. Source code analysis is limited to scanning uncompiled code only. The first tools to reach the market\(^3\) were static in their approach to finding vulnerabilities, by simply searching for usage of function calls with known possible vulnerabilities. These kinds of tools does not require much processing power and are in general fast to run but result in a high amount of false positives\(^9\). Recent

\(^2\)Custom developed code refer to all code that is written in-house and is specific to the application. Such source code will always be available to the project.

\(^3\)Tools such as ITS4\(^{30}\) Flawfinder\(^{17}\) and RATS\(^{50}\)
tools use more advanced testing techniques, described in section 4.3.2, making them more thorough and precise but also way more time-consuming. For static code analysis, the scope is limited to code that is available in source code form, which restricts the analysis of third party libraries and frameworks to those that are open source and written in a language supported by the tool.

4.3.1.2 Binary Analysis

Binary analysis is similar to source code analysis as it uses the same techniques to find vulnerabilities, but with binaries instead of source code. Analysing the binaries or bytecode increases the testing scope as it allows analysing code that is linked from external components[4] and can also be considerably faster[39]. A drawback however when using binary analysis is that the code has to compile in order to be analyzed, because otherwise it can’t be translated to proper machine code.

4.3.1.3 Bytecode Analysis

Bytecode analysis also uses the same techniques to perform analysis, but with bytecode as a source instead of source code or binaries. Bytecode is a form of instruction set that lies between source code and machine code. Languages that uses virtual machines (VM) to run, such as Java and C#, compiles to bytecode when built, and is then compiled into machine code by the VM just before execution. Bytecode analysis also achieves a wider scope than source code analysis, but just like binary analysis it requires code to compile. Bytecode analysis also has a wider scope than source code analysis, as it can analyze both the source code as well as third party libraries that are pre-compiled to bytecode and open source libraries written in languages compiled to bytecode.

It should be noted that binary and bytecode analysis is assumed to be used in projects that use development techniques that makes sense with the chosen testing technique. For example, binary analysis should be used in projects developed in languages that is compiled to machine code binaries, while bytecode analysis is appropriate for systems developed to run on virtual machines\(^4\).

4.3.2 Underlying Static Analysis Techniques

SAST applies various techniques for statically analyzing applications to detect vulnerabilities. These techniques are often derived from compiler technologies.

4.3.2.1 Control Flow Analysis

Control flow analysis is a technique that determines the control flow of a program by constructing a control flow graph, as illustrated in Figure 4.4.

\(^4\)Java VM, .NET CLR etc.
4.3.2.2 Data Flow Analysis

Data flow analysis is a technique that gathers information about possible set of values for variables at various points in applications. The points are constructed with control flow analysis, and results it different states.

4.3.2.3 Taint Analysis

Taint analysis utilizes results from control and data flow analysis to detect vulnerabilities. It’s a form of information-flow analysis that establishes whether values from untrusted methods and parameter may flow into security-sensitive operations by using sources, sinks and sanitizers. Sources are methods whose return values are considered tainted or untrusted. Sanitizers are methods that manipulate untrusted data to produce taint-free output. Sinks are methods that in some way perform security-sensitive computations. From this information security rules are constructed, stating combinations of sources and sinks that can be vulnerable and which sanitizers that should be used to make sure that the combination of sources and sinks are safe.

In Figure 4.5, the second step is where models of applications is built by data and control flow analysis. The result is then utilized in the analysis phase, where it’s combined with security knowledge in the form of rules for taint analysis.
4.3. STATIC APPLICATION SECURITY TESTING

4.3.2.4 Configuration Analysis

Configuration analysis performs analysis by scanning an application’s configuration files, host settings or server configurations. Configuration analysis mainly provides assurance that the application operates within its desired security context, therefore being most useful when evaluating security late in development processes.

4.3.2.5 Dependency Analysis

According to Aspect Security[60], 26% of all downloaded libraries in their study had known vulnerabilities. Dependency analysis is a technique that scans for the use of external dependencies, and runs them by a blacklist with components that have known vulnerabilities. The technique is easy to run and implement, but requires the blacklist to be continuously kept up to date with vulnerable components to be useful.

4.3.3 Detection of Vulnerabilities

By using control flow and data flow analysis to build a simplified model of the application, SAST tools can apply configuration, dependency and taint analysis to analyze systems for vulnerabilities.

FindBugs[16] is an open source SAST tool for Java that categorizes its possible findings into nine groups: bad practices, correctness, experimental, internationalization, malicious code vulnerabilities, multithread correctness, performance, security and dodgy code. Security is the group related to web application vulnerabilities and contains the possible findings displayed in the list below. The list is obviously not representative for all SAST tools, but it provides a notion of what’s possible to detect.

- Hardcoded constant database password
- Empty database password
- HTTP cookie formed from untrusted input
- HTTP response splitting vulnerability
4.3. STATIC APPLICATION SECURITY TESTING

- Absolute path traversal in servlet
- Relative path traversal in servlet
- Non-constant string passed to execute method on SQL statement
- A prepared statement is generated from a non-constant string
- JSP reflected cross site scripting vulnerability
- Servlet reflected cross site scripting vulnerability in error page
- Servlet reflected cross site scripting vulnerability

The main drawback of SAST tools is they cannot be sure of how an application will execute and therefore they have to make assumptions. This leads to the problem of false positives and negatives.

4.3.4 False Positives vs. False Negatives

SAST often produce false positives results, which are reported issues that are not actual problems (Figure 4.6). This can occur for several reasons, but it often boils down to static tools not being sure of the integrity and security of data as it flows through applications. The main reason is that SAST has to make assumptions about how systems actually function in order to perform analysis. For example, if closed source components or external systems are used, it may be impossible to trace the flow of data using SAST.[46]

The precision of analysis determines how often false positives are reported; the more imprecise, the more false positives. Unfortunately, precision usually depends on analysis time. The more precise the analysis is, the more resource consuming it will be and the longer time it will take. This implies that precision has to be traded for time of analysis. If the analysis is fast, it is likely to report many false positives, making analysis unreliable. On the other hand, a very precise analysis is unlikely to terminate in reasonable time for large programs.

One way of reducing false positives is to filter the result and remove potential errors that are unlikely. The problem with such approach is that it results in removal of positives that are indeed defects. This is called a false negative (Figure 4.6), i.e an actual problem that is not reported. There exist several well-established techniques that can be used to trade-off precision with time of analysis, such as making analysis flow-sensitive, path-sensitive or intra-procedural[13]. There is no simple answer to which approach that should be used to resolve the dilemma of false positives contra false negatives and different vendors practices their own distinct solutions.
4.4 Dynamic Application Security Testing

DAST, or dynamic testing/scanning, is often referred to as black box testing since it tests applications from the outside. DAST has no information about what happens inside of applications and always requires systems, or at least parts of them, to actually execute\(^5\).

\(^5\)This implies that versions that do not build correctly or that for some other reason do not run cannot be analyzed by DAST.

![Figure 4.6: False positives and negatives.](image)

4.4.1 Testing Scope

As illustrated Figure 4.8, the scope of DAST can reach across all modules of a system as they all interact and are executed. However, as DAST does not know the inner structures of applications, it has difficulties covering all segments of each component. Thus, the possible

![Figure 4.7: Dynamic analysis tests from the outside.](image)
testing scope is narrower for each section than for SAST, but reaches across all different layers.

4.4.2 Penetration Testing

The term penetration testing is somewhat used in different manner in various literature. For this thesis, penetration testing is defined as a technique that finds vulnerabilities by trying to attack systems by acting as an outside user. Penetration testing is the only existing security testing technique that actually tries to break applications by performing real attacks and analyzing systems’ responses. The *OWASP Testing Guide*[^45] describes penetration testing as follows.

> *The ‘art’ of testing a running application remotely, without knowing the inner workings of the application itself, to find security vulnerabilities*  

Technically, penetration testing consists of a combination various techniques. It begins with scanning, or mapping, applications to identify their attack surface by constructing a structure with possible input sources. The result from scanning is then utilized in different ways to find cracks and vulnerabilities in systems, usually by sending modified requests and analyzing the results. Doing so triggers systems to execute, but does not provide any information about what is actually happening at the inside. Thus, penetration testing tools have a limited scope as they does not know which parts of systems that were tested, and how much coverage that was achieved (Figure 4.8).
To describe a complete penetration testing process, it can be categorized into four groups\(^6\): Crawlers, Proxies, Vulnerability Scanners and Fuzzers. The different techniques can be used independently, but a mixture is required to perform a complete penetration test that both scans and analyzes applications in order to detect vulnerabilities. To understand the technical approaches that are used, these techniques are explained independent of each other, but, as mentioned, a combination is assumed when performing a complete penetration test. Figure 4.9 illustrates how the different components in a penetration testing process are combined.

Figure 4.9: Illustration of techniques used in penetration testing.

As penetration testing is a DAST technique that applies black-box testing, it only sees applications from the outside and therefore somehow has to gain information about its inner structure and attack surface. Running web application spiders does this and it’s the first step of a penetration test. The result from running a spider is a map over applications different pages and entry points. Vulnerability Scanners and Fuzzers are techniques that utilize the generated attack surface from spiders, to find weaknesses of an application. These techniques are the two most common in penetration testing, but are sometimes extended with various methods\(^7\) to increase the range of possible findings of vulnerabilities.

\(^6\)The categories are from Kali, previously called backtrack, which is a Linux distribution used for penetration testing, which comes preinstalled with numerous penetration testing programs.

\(^7\)An example of an extension of techniques to find vulnerabilities is a Session Token Analyzers, whose
The amount of network traffic consumed and time spent when performing penetration tests vary a lot depending on the tools used. A study made in 2010 at Stanford University[31] compared different penetration testing tools. It indicated that the execution time between the fastest and slowest tools differed by over 700% (Figure 4.10). It also indicated that the amount of data sent differed by almost 500% (Figure 4.11) and data received with more than 1600% (Figure 4.11). As further discussed in the analysis section, the difference between tools has to be taken into consideration when choosing a penetration testing tool.

![Figure 4.10: Comparison of testing time for various DAST tools[31].](image)

![Figure 4.11: Comparison of network traffic used for various DAST tools[31].](image)

It is often argued that fully automated penetration tests are limited in their findings of vulnerabilities compared to manual approaches. The main argument is that tools are too generic, meaning that they are not designed for applications' custom code[45]. They do possess enough knowledge to find nonspecific problems, but flaws that are deeply intertwined purpose is to analyze the randomness properties of tokens, such as session cookies.
4.4. DYNAMIC APPLICATION SECURITY TESTING

in business logic and custom application design gets disregarded. There is obviously some
truth in such statements, but to what extent a fully automated penetration test and manual
one differs is outside of this thesis' scope and should be further studied.

4.4.2.1 Web Application Spiders

The goal of web application spiders is to construct a map over an application’s structure and
attack surface. Web application spiders work in similar ways as web spiders by recursively
requesting web pages and parsing them for links to other content, fetching these pages from
the new content and then parsing them until no new information can be gathered.

As modern web applications are dynamic and changes appearances depending on who uses
them and when, spiders have to address various challenges that complicates the mapping
process. The Web Application Hacker’s handbook[55] identifies the following problems that
spiders have to address:

- Form-based navigation using drop-down lists or text input.
- JavaScript-based navigation, such as dynamically generated menus.
- Links buried in compiled client-side objects such as Java or Flash.
- Multistage functions, requiring several steps to be performed in a defined sequence.
  For example, logging in as a user requires filling in a registration form, which will
  probably be neglected when wrong input is passed. If the spider is not smart enough
  to fill in correct information, testing the application with user privileges is not possible.
- Handling authentication and sessions. The spider has to be manually configured with
  required authentication tokens in oder to view privileged pages. However, even if this
  is done correctly, it may result in further problems;
  - By following all URLs, the spider will probably logout, causing the session to
    break.
  - If the spider provides invalid input to sensitive functions, an application may
    defensively break the session.

Resolving such problems is not an easy task, making automated spiders inaccurate. One
usual way of resolving these problems is to extend the automated spiders with manual map-
ning of applications. This can for example be performed by positioning a proxy between a
client’s interface and a system and then capturing all data being sent. When the application
then is manually used, the proxy captures requests and responses that are used to build a
structured map over the system.

Spiders can also be extended to find hidden content by using brute force techniques. This
can help the finding of names or identifiers by making huge numbers of requests to the web
server with more or less educated guesses, but is usually done by iterating through lists of
common directory names and capturing details of the server response.

As the result of running an automated spider is a map over applications, it cannot reveal vul-
nerialities itself. Doing so requires additional tools and as discussed earlier, a penetration
test requires combination of different techniques.
4.4.2.2 Proxies

Proxies as security testing tools are connected between the client’s browser and the web application and is therefore able to see everything sent over the network when clients use a system. It’s just as well possible to construct penetration testing tools that are not proxies, although those that are proxies gain benefits that otherwise are difficult achieving.

As proxies works as a man-in-the-middle, they enable bypassing of all client-side validations and selective modification of network messages that are sent. Further, they can be used to handle encrypted communication, such as SSL, by creating self-signed certificates that the client trusts while the proxy itself uses the applications' certificate to talk with the application. Figure 4.12 illustrates how this is done in the Zed Attack Proxy, ZAP\(^8\).

![SSL with ZAP][61]

4.4.2.3 Vulnerability Scanners

The purpose of vulnerability scanners is to find vulnerabilities in web applications by either manipulating HTTP messages or inspecting them for suspicious attributes\(^{15}\). The tools look for vulnerabilities either passively or actively and use several different techniques to attack applications. Active vulnerability scanners actually attack applications by constructing and sending forget requests and detect weaknesses by analyzing the responses. Active vulnerability scanners needs to be initiated in order to run and mainly looks for vulnerabilities that requires interaction between users and applications, such as XSS, SQLi, HTTP header injection and file path traversal. Passive vulnerability scanning on the other hand does not actively run applications, although it still requires active scanning or spidering in order to gather a set of requests and responses to be analyzed. Passive scanning focuses on finding vulnerabilities that does not require interaction with systems, such as clear text credential submission, cookie misconfigurations and so on.

In order for web applications to provide specific access levels to different users, web applications employ various authentication and authorization mechanisms. A vulnerability scanner must be able to login to applications and to manage sessions by not happening to log out, or at least knowing when if it was logged out, to be able to test applications.

Several different categories of common vulnerabilities, with a fairly standard signature, are possible to detect by scanners with a certain degree of reliability\(^{55}\). The signature is

---

\(^8\)Zed Attack Proxy, ZAP, is a penetration testing proxy tool developed by OWASP\(^{61}\).
some indication of vulnerable behavior in the response and can either be found in normal communication or by crafted requests designed to trigger the signature if the vulnerability exists.

Following is a list with examples of vulnerabilities that are possible finding using vulnerability scanners\[55\]. The list is not representative for all existing vulnerability scanners, but aims to provide an idea of what is possible finding using penetration testing techniques.

- Reflected cross-site scripting, XSS.
- SQL-injections (those resulting in error messages)
- Path traversal vulnerabilities
- Command injection
- Straightforward directory listings
- Sending of sensitive data, such as passwords, in clear text
- Accessing unauthorized content

It is important to notice that vulnerability scanners cannot reliably detect all instances of vulnerabilities of the same category using a standard attack string and signature. For example, an application might use some level of input validation that is able to block or sanitize the crafted inputs comming from a scanner. However, such validation might still be bypassed by skilled attackers that use techniques that the tools does not know of. Furthermore, some attacks such as SQL injections or stored XSS does not result in error messages in the response, which makes those vulnerabilities more difficult detecting with vulnerability scanning.

4.4.2.4 Fuzzers

Fuzzing is the process of creating and sending malformed data as input to applications and analyzing responses. If an application fails unexpectedly or if the response diverges from standard answers, a bug and a potential vulnerability has been found. The goal of fuzzers is to exercise code that analyzes data structures, loosely referred to as parsers.

Parsers should handle all data received from clients in web applications. This can for example be data such as navigation in the UI, input fields and file-inclusions. Fuzzing these parsers is done by sending data that applications do not expect, such as too large amounts, negative integers or unfamiliar symbols. By testing as many combinations as possible in carefully thought-out ways, weak parsers might break, which often indicates vulnerabilities in a system. Fuzzers can either work by malformed data randomly or running through a list of known possible sets of malicious data.

Fuzzers are similar to active vulnerability scanners in their technical approaches. Both methods send requests to applications and analyze responses. The difference lies in that fuzzers uses a more trivial and brute-forcing approach, where the next request sent does not depend on prior responses. Vulnerability scanners tests a wider scope of attacks and can apply logic to how testing is performed. Depending on the response from a crafted request, vulnerability scanners can decide to dig deeper in a specific hole or try other approaches towards breaking an application.
4.5. INTERACTIVE APPLICATION SECURITY TESTING

Fuzzers can identify flaws or bugs by analyzing all entry points of applications, where user input is sent to an application. If a parser or validator is completely missing at such entry point, it’s a design flaw, and if the parser just does not function correctly it’s a bug in the implementation of the parser. The scope of possible findings when using fuzzing is limited to vulnerabilities that are related to incorrect or missing input validation.

4.5 Interactive Application Security Testing

Interactive application security testing (IAST) is a relatively new method of security testing for applications, with the first tools appearing only a few years ago[40]. The term IAST was firstly introduced by Gartner and has since been accepted and used by several vendors of IAST tools. IAST refers to all grey box\(^9\) security testing methods that in different ways enables testing and analyzing applications with knowledge of both inner and external structures.

4.5.1 Testing Scope

IAST uses grey-box testing and has knowledge of both inner and external structures of applications. This combination enables a wider possible test coverage than for SAST or DAST techniques. The knowledge about systems’ inner structures can help provide information about which code sections that has been analyzed, making it possible to analyze all code that can be accessed and thus resulting in a wide coverage within each component. Having knowledge of external structures implies that the applications are running, which enables IAST to also test the interaction between different components. The possible scope is illustrated in Figure 4.13. However, it should be noted that using IAST only enables the possibility of acquiring a comprehensive testing scope, but it is not in any way guaranteed that such is the case for all IAST tools. How the actual scope looks like depends on specific techniques and implementations.

Three different techniques using IAST has been identified, and are further discussed here.

---

\(^9\)Grey box testing is a combination of white and black box testing, as it has knowledge about both the inside as well as the outside of applications.
4.5. INTERACTIVE APPLICATION SECURITY TESTING

4.5.2 Runtime analysis

Runtime Analysis, RA, monitors the behavior of an application at runtime. There are different approaches towards achieving runtime monitoring, where the most commonly used and therefore discussed here is *instrumentation*. Instrumentation is a technique that exists within most programming languages using virtual machines. For Java, the instrumentation library was introduced in version 1.5 released 2004[32] and has thus existed for several years. When first released, it was mostly used for performance measuring and debugging, but around 2011/2012, tools such as Seeker and Contrast that uses runtime analysis to find vulnerabilities reached the market. Instrumentation enables injection of bytecode into an application after it has been compiled. Doing so, RA can monitor specific components and APIs\(^\text{10}\) passively without affecting systems behavior at runtime.

Runtime analysis uses a so called IAST agent, or engine, to monitor all calls to given APIs in runtime and to track data by tagging it as it enters, exits and flows through systems. The inserted bytecode is referred to as passive sensors and are injected across the entire application stack, as illustrated in Figure 4.14. As the sensors are passive, runtime analysis is dependent on the code to execute and the system to be used by some external tool. There are various ways to automatically use systems, for example by automated functional integration or system testing.

Using the instrumentation technique, runtime analysis tools can track the flow of data

\(^\text{10}\)Runtime analysis can for example monitor End-to-end HTTP requests, database and directory calls, file system calls, string manipulations, memory manipulations, usage of third party libraries, calls to external applications, APIs and so on...
through applications by tagging data as tracked when it arrives as input, and then follow the
data as it progresses through different sections of an application (Illustrated in Figure 4.15).
If data from a sensitive source is used when constructing an SQL statement, without being
validated first, a possible SQL injection is detected. This process is similar to SAST using
taint analysis, but obtains less false positives as it does not require making any assumptions
of how the code functions. In further similarities with SAST, runtime analysis can access
configuration files, and thus use configuration analysis as described for SAST (section 4.3.2.4
at page 35). For dependencies, runtime analysis can both use the technique for dependency
analysis as for SAST (section 4.3.2.5 at page 35), as well as scanning the actual code of
libraries that runs on the same virtual machine as the IAST agent.

Figure 4.15: Runtime analysis tracks data by tagging it[29]

Runtime analysis only consists of components that lie inside of applications, but is neverthe-
less thought of as a technique that also has knowledge of applications' external structures.
This relies in the fact that tracking real data in runtime allows IAST agents to see the exact
external requests that was made by clients when they use an application. Thus, using only
a runtime analysis tool cannot control how systems are executed, but when data enters a
system, runtime analysis obtains knowledge of the exact structures of the network traffic
that entered.
4.5. INTERACTIVE APPLICATION SECURITY TESTING

4.5.3 Combination of DAST and Runtime analysis

Runtime analysis is a passive form of security testing and thus requires additional external tools that actively use running applications. Combining runtime analysis with DAST tools is one way of achieving the required combination, and it’s an IAST approach used by market leaders in the security testing field[27]. As illustrated in Figure 4.16, the dynamic scanner tests applications from the outside, while the applications are being monitored by an IAST agent, or runtime analysis agent, at the inside. The agent can provide DAST tools with information about how the scanning should proceed and also possess information about internal state that otherwise would be inaccessible for DAST. Following are examples of information that can be accessed by the dynamic scanner at runtime[58]:

- Web app runtime activities
- Application structure, environment, technology, components
- Configuration files
- Source code information
- Log files
- File-system activities
- Registry accesses
- Network traffic
- Database access

Figure 4.16: Combination of DAST and Runtime Analysis[59]

By receiving this information, several drawbacks of DAST are eliminated or at least reduced. Running dynamic security testing usually result in narrow and undetermined code coverage. IAST agents can provide inside information to the external DAST tools about which sections of the application that has been analyzed, and most importantly, which hasn’t. Furthermore, IAST agents also extend the range of possible findings of vulnerabilities, by for example finding hidden parameters, backdoors and non-reflected injection issues as well as enabling access to otherwise unreachable internal files.
4.5. INTERACTIVE APPLICATION SECURITY TESTING

4.5.4 Combination of SAST and DAST

The output from static analysis can be used to create and tune the dynamic testing[27]. A combination of the two enables reduction of the main drawbacks of both SAST and DAST. SAST can possibly result in a high degree of false positives, which is resolved with running actual dynamic testing to verify the vulnerabilities. Furthermore, the main disadvantage with DAST is the absence of being able to pinpoint exactly where in the code the actual problem lies, which also gets resolved using this combination.

The combination of SAST and DAST is an interesting solution, and several studies and presentations propose solutions to how the two techniques can be combined[47] [53] [52] [1]. However, no market tools using the technique were found during the writing of this thesis, but there seem to exist PoC and academic implementations. For this reason, the technique is disregarded in the analysis.
Chapter 5

Analysis

The analysis is split into two sections; the first evaluates how well the identified security application testing techniques fit into continuous integration environments and the second section focuses on which kinds of security risks and requirements the testing techniques can assure.

The identified security testing techniques that were analyzed are SAST – Static Application Security Testing, DAST – Dynamic Application Security Testing and IAST – Interactive Application Security Testing. For SAST and DAST, the properties are mostly evaluated at a high level. However, the underlying techniques are exposed and evaluated separately when necessary. The underlying techniques of IAST, runtime analysis and DAST + RA, use completely different techniques in their way of analyzing systems, and are therefore evaluated separately throughout the entire analysis section.

The purpose of the analysis is not to identify one technique that always is most suitable for usage in continuous integration, but rather to highlight good and bad qualities for each technique.

5.1 Evaluation of AST in Continuous Integration

The analysis uses the properties described in section 3.7 at page 22. The properties are evaluated one at a time for each security testing technique and each section wraps up with both a summary as well as an estimated score of how well the different security testing techniques satisfies the property.

There exist several distinct implementations from different vendors of all the security testing techniques discussed in this report. These implementations might use different solutions and approaches and therefore behave somewhat differently, with various accuracy. The term specific tools refer to a specific implementation of the testing technique. Which specific tool to use is outside the scope of this thesis, but guidelines on qualities to look for are provided.

\[1^1\text{A high level means that it's enough information for the evaluation to know that SAST tests statically from the inside, while DAST performs analysis dynamically from the outside.}\]
5.1. EVALUATION OF AST IN CONTINUOUS INTEGRATION

5.1.1 Automated execution

More thoroughly explained rationale and criteria for automated execution is found under section 3.7.1 at page 22.

Automated execution is possible if the whole testing process, except from initiation, can run automatically. The evaluation of this property is based on if automated execution is possible at all and also if there are any potential drawbacks of not performing tests manually.

5.1.1.1 SAST

SAST tools are not intended for manual execution. Full automation is possible.

5.1.1.2 DAST

Most penetration testing tools can run in either automated or manual mode, making full automation possible with the latter option. As discussed in section 4.4.2 at page 38, there are several drawbacks of fully automated penetration tests and many argue that using full automation only enables finding the most basic vulnerabilities. To reach the full potential of penetration testing, human knowledge is required. To some degree, this can be managed by configuring and enhancing tools before testing, by for example manually mapping applications instead of letting spiders doing so. This resolves problems with session management and unsupported protocols that spiders usually struggle with. Fuzzers and vulnerability scanners can then detect weaknesses by using the manually mapped structure.

5.1.1.3 Runtime Analysis

Runtime analysis differs from both SAST and DAST, as it conducts testing passively. As runtime analysis analyzes applications passively it does not provide the option of automated execution itself, but rather requires external arrangements to actually execute applications. The property of automated execution therefore depends on an external tool that is used to run runtime analysis automatically. However, it should be noted that one of the core concepts of continuous integration is to continuously test that all components of an application are working and integrate. If such practices exists within a project, automated integration and system testing is most likely already implemented, which is enough for runtime analysis to analyze a system.

5.1.1.4 DAST + Runtime Analysis

The combination of DAST and runtime analysis is a good way of automating runtime analysis, as the DAST tool will work as the external component that runtime analysis requires. Furthermore, runtime analysis also leads to an increased base of knowledge about the inner structures of applications that DAST tools otherwise lack, probably making it more comprehensive and less dependent of human knowledge.

5.1.1.5 Summary

SAST is the technique that best supports full automation on its own. Although, for projects where integration and system testing is performed, IAST can be thought of as equally automated. As the philosophy of DAST is to act as a real threat, by attacking applications
from the outside, the usefulness of full automation is often argued to be limited. Table 5.1 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td>DAST</td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td>RA</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Estimated score of the automated execution property

5.1.2 Headless

More thoroughly explained rationale and criteria for headless execution is found under section 3.7.2 at page 22.

Headless execution refers to the possibility of initiating testing from a command line interface or via an API. This property is important to enable integration in CI environments at a technical level, as graphical interactions in general are difficult to automate. If a tool provides headless execution or not often depends on specific implementations, but some general concepts can be deduced at a higher level, which is further discussed here.

5.1.2.1 SAST

SAST uses active analysis methods and therefore requires to be initiated in order to run. As static analysis methods are built for completely automated testing processes, there is no real reason for tools to use graphical interfaces. There exist several tools that run from the command line, which enables the possibility of headless execution of SAST.

5.1.2.2 DAST

Penetration testing is an active testing method that needs to be initiated with an application’s structure and suitable attacks. As the technique is often considered to require manual steps, many modern tools use a graphical interface to interact with users, but also often include plugins that allow command line execution.

5.1.2.3 Runtime Analysis

Runtime analysis tools use the instrumentation technique and are integrated as standalone components before systems are compiled. Analysis is performed passively by analyzing systems from the inside as they get executed and does not itself require initiation to begin. However, the technique requires a backend component to store findings. This is the module that the CI system needs to communicate with and that has to be available headlessly through a command line interface. Several of the existing runtime analysis tools provide APIs.

---

2Tools might use graphical interfaces to present findings, but does not require any interfaces while it performs testing.

3Contrast and Seeker are examples of tools providing APIs to supply results.
5.1.2.4 DAST + Runtime Analysis
As for DAST, the dynamic tool has to be initiated through a command line interface.

5.1.2.5 Summary
As DAST is often used manually and interactively, graphical interfaces are common. When choosing tools, one has to make sure that both initiation and reporting is possible to perform through a command line interface. SAST and runtime analysis tools are not intended for manual usage while testing, as they do not require interaction with developers while testing. SAST and runtime analysis are well suited for command line interactions, while DAST or DAST + runtime analysis might support it. Table 5.2 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td>DAST</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Estimated score of the headless property

5.1.3 Repeatable
More thoroughly explained rationale and criteria for repeatable execution is found under section 3.7.3 at page 22.
To acquire proper regression testing, security testing has to be repeatable. Tests have to use deterministic algorithms, without randomness and with a repeatable database state to accomplish this property.

5.1.3.1 SAST
As none of the SAST techniques execute code, there is no state that has to be kept intact between test runs. Usage of randomness depend on specific implementations, but there is no obvious use case where it is required. SAST tools are most likely deterministic, but specific tools should be evaluated for this property.

5.1.3.2 DAST
Web application spiders and vulnerability scanners use deterministic algorithms that are repeatable. Fuzzers can either use lists of known vulnerabilities or randomly generated data to perform testing. To fit into CI environments, the deterministic approach with pre-defined lists should be used.
Dynamic security testing will most likely utilize the backend database when performing tests, but usually doesn’t depend on or modify the data that the database consists of. Although, when detecting some types of injection vulnerabilities, such as stored SQLi and XSS, it might inject persistent data into backend databases. Testing for those vulnerabilities dynamically has to be carefully controlled, probably by restoring the database afterwards.
5.1.3.3 Runtime Analysis
Runtime analysis passively analyze data as it progress through systems in runtime. If the same data flows in equivalent ways, testing will be repeatable. In other words, if the component that triggers system execution is repeatable; then runtime analysis also is. As for DAST, applications’ databases will most likely be used, and for systems to execute equivalently between different runs the state of the database should remain unchanged.

5.1.3.4 DAST + Runtime Analysis
Runtime analysis will depend on the active testing from the DAST tool. The repeatability of DAST + Runtime Analysis therefore depends on the repeatability of DAST.

5.1.3.5 Summary
As static analysis techniques are the only methods that do not execute systems, it is best suited to acquire repeatability. Both DAST and RA execute systems when performing analysis, which can compromise the repeatability if the database state is modified. A modified database does not guarantee the loss of repeatability, but this fact should at least be regarded when using dynamic and interactive testing methods. Table 5.3 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Estimated score of the repeatable property

5.1.4 Internal expertise
More thoroughly explained rationale and criteria for internal expertise is found under section 3.7.4 at page 23.

Utilizing development teams’ in-house knowledge is an important agile quality, stated in the agile manifesto (section 3.2 at page 17). To acquire this property, external expertise has to be excluded by assembling required security knowledge into the testing tools. Doing so, the tools has to be able to provide distinct information about how reported vulnerabilities can be resolved, obtain a low rate of false positives and require none or a low amount of fine-tuning.

5.1.4.1 SAST
SAST has knowledge about systems’ actual sources, which enables tools to determine exact positions of vulnerabilities in code and therefore also how they can be can be resolved. On the other hand, it does not know what the possible vulnerability looks like from the outside
and cannot provide information about how it can be exploited. Such information is useful when resolving false positives.

SAST tools often struggle with false positives, false negatives, or both. The balance between false positives and false negatives depends on how specific tools are implemented, which has to be regarded when choosing tools (further discussed in section 4.3 at page 31). Determining whether a reporting is correct or not often requires security expertise, which is why tools with low rate of false positives should be chosen in order to obtain the internal expertise property.

Furthermore, tools often have to be fine-tuned with explicit rules to reduce the amount of false positives and to expand the scope of possible findings. How much work this calls for also depend on specific tool implementations. Tools with a wide range of possible findings and low amount of false positives require less fine-tuning.

5.1.4.2 DAST

DAST, compared to SAST, tests running systems, which theoretically lowers the amount of false positives as it does not require making assumptions about how data flows. However, as penetration testing use black-box techniques, it does not possess information about the position in code where vulnerabilities exists. Tools can only present how vulnerabilities look from the outside and how they might be exploited. Locating vulnerabilities in the code can be a difficult and time-consuming process.

5.1.4.3 Runtime Analysis

As runtime analysis knows about actual data as it flows through a system, it has the capacity of both knowing how a vulnerability can be triggered from the outside as well as where it exists in code. Thus, runtime analysis has the potential of providing information about how to exploit a vulnerability and how it should be fixed, thus reducing the need for external expertise.

5.1.4.4 DAST + Runtime Analysis

The combination of DAST and runtime analysis also obtains knowledge about both the inside and outside structures of systems, and is therefore able to provide information about how vulnerabilities can be exploited as well as where they exist in the code. It should also be able to lower the rate of false positives compared to only using runtime analysis, as vulnerabilities are detected from multiple angles and therefore with a higher level of certainty.

5.1.4.5 Summary

SAST has the drawback of not knowing how vulnerabilities can be exploited and can also include a high amount of false positives in its reports. DAST does know how findings can be exploited, but instead does not have the knowledge of knowing which code is vulnerable and how the vulnerability might be resolved. Runtime analysis on the other hand, both has knowledge of internal data and external triggers and is therefore able to provide information about both the location of vulnerable code and how vulnerabilities might be exploited. However, the combination of DAST and runtime analysis obtains the same information
5.1. EVALUATION OF AST IN CONTINUOUS INTEGRATION

as runtime analysis but with a possibly lower rate of false positives. Table 5.4 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td></td>
<td>SAST</td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td>DAST</td>
</tr>
</tbody>
</table>

Table 5.4: Estimated score of the *internal expertise* property

5.1.5 Automated decisions

More thoroughly explained rationale and criteria for automated decisions is found under section 3.7.5 at page 23.

Being able to automate decisions regarding whether a system is secure or not is required when obtaining a full chain of continuous delivery, which often is the final goal of continuous integration. To enable automated decisions, security tests have to be trustworthy by obtaining a high test coverage and preferably provide information about which sections of the code that were tested and not. The reports also have to be reliable in the sense of not including, or at least having a low amount of, false negatives.

5.1.5.1 SAST

The trustworthiness of SAST can be limited, due to results either containing false positives or false negatives. Tools with low levels of false negatives should be chosen to acquire this property.

As discussed previously, SAST tools have the feature of knowing which parts of the code that has and has not been analyzed. However, the testing scope differs depending on specific SAST techniques. Source code analysis can analyze the entire code base of both internal code and third party dependencies that are open source, but fails to analyze compiled libraries. Binary and bytecode analysis has a wider scope than source code analysis, as it can analyze compiled libraries as well (testing scope of the different SAST techniques is illustrated in Figure 4.3).

5.1.5.2 DAST

As DAST tests systems from the outside, it cannot gain information about which parts that were executed and not. This can lead to false negatives, as the sections that are not tested might contain vulnerabilities that then get disregarded. As DAST runs the complete system it enables reaching all layers of systems, such as third party libraries and runtime environment configurations, but the scope is still limited to what spiders finds (illustrated in Figure 4.8). Semi-automated solutions, where the scope is manually controlled once and then rerun could be a possible solution to widen the testing scope of DAST.
5.1.5.3 Runtime Analysis

The scope that is tested in runtime analysis is directly related to test coverage of external components that execute systems. Measurements of standard test code coverage can thus be a helpful variable to use when constructing rules for automated decisions. As runtime analysis both has knowledge about what is tested and how data internally flows, it has the possibility of acquiring a wide testing scope with a low level of false negatives. However the scope has to be manually controlled with the external component, which is a drawback of runtime analysis.

5.1.5.4 DAST + Runtime Analysis

When combining DAST and runtime analysis, drawbacks of both methods are reduced. The DAST tool will obtain knowledge of what parts of the system that is tested and therefore try to increase the testing scope with additional tests. The automated dynamic tool controls the scope for runtime analysis automatically, which otherwise has to be performed manually. DAST + Runtime analysis should be able to obtain a wide scope with a low level of false negatives\(^4\).

5.1.5.5 Summary

SAST has knowledge about which parts of systems that are analyzed, but the testing scope is limited when it comes to third party code and results might comprise false negatives. DAST on the other hand can reach all third party components, but fails to identify which parts of systems that gets analyzed and has a scope that is limited to findings of spiders. The testing scope of runtime analysis depends on the test coverage of external tools, which is usually controlled manually. DAST + runtime analysis however also has information about the test coverage but should be able to acquire fewer false negatives than the other testing techniques. Table 5.5 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td></td>
<td>RA</td>
<td>SAST</td>
</tr>
<tr>
<td></td>
<td>DAST</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Estimated score of the *automated decisions* property

5.1.6 Fast execution

More thoroughly explained rationale and criteria for fast execution is found under section 3.7.6 at page 24.

For security tests to be suitable in rapid agile projects, tests should be fast so that they don’t affect the development process in too large extent.

---

\(^4\)False negatives stem from either not testing parts of the code or by trying to remove false positives and thereby mistakenly disregarding a real vulnerability.
5.1.6.1 SAST
Static code analysis can be very time-consuming for large systems. Some tools enable scanning of only newly committed code, which then can be run for each build, while scanning of the whole code base only is performed nightly.

5.1.6.2 DAST
As spiders, active vulnerability scanners and fuzzers require data to be sent back and forth, a possibly large amount of network traffic is sent when performing penetration tests. This issue can partially be resolved by running less precise testing daily and exhaustive testing at nightly builds or by making sure that the test tool runs at the same physical machine as the application.

The amount of traffic sent as well as the time of analysis for dynamic testing depends heavily on which specific tool that is used. The fastest tool with least amount of network traffic required is a valid option for this property.

5.1.6.3 Runtime Analysis
Runtime analysis does not require conducting any explicit testing, but instead depends on integration and system testing to run. Data has to be analyzed and processed as systems run, which does slow down the process to some extent. Still, as RA does not require its own testing process, it is quicker than both SAST and DAST.

5.1.6.4 DAST + Runtime Analysis
The execution speed is similar to DAST tools but possibly a bit slower since runtime analysis slows down applications it analyze. As for DAST and SAST, it has to be performed in addition to functional tests.

5.1.6.5 Summary
Both SAST, DAST and the combination of DAST + runtime analysis runs in addition to functional testing and can be too time-consuming to run with each build. Only running small sections of the testing scope daily and the whole system nightly can resolve this issue. Runtime analysis on its own passively analyze the system during external tests. This slows down the testing process to some degree, but is still faster than the other techniques.

It should also be noted that the testing speed depends on the implementation of specific tools. Tools that are considered fast should be chosen to obtain this property. Table 5.6 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>SAST</td>
<td>DAST</td>
<td>DAST + RA</td>
</tr>
</tbody>
</table>

Table 5.6: Estimated score of the fast execution property
5.1.7 Early testing

More thoroughly explained rationale and criteria for Early testing is found under section 3.7.7 at page 25.

The main purpose of integrating security testing into continuous integration is to build in security throughout the complete development processes. This means that security testing should be integrated as early as possible.

5.1.7.1 SAST

SAST does not require the code to neither build nor execute and is therefore possible to integrate early and continuously in the SDLC.

5.1.7.2 DAST

DAST requires a running web application to perform testing. Depending on the software being developed, the stage when a somewhat working system exists will differ, which might restrict the usefulness of DAST in CI if integrated late. Furthermore, in builds where the code does not compile or run correctly, dynamic testing is not possible.

However, it should be noted that agile development strives towards early and continuous delivery of valuable software, which probably results in a relatively early and continually working system and thus correspondingly DAST testing.

5.1.7.3 Runtime Analysis

The runtime analysis component can be integrated from the very beginning of development, but will initiate testing when applications start running and functional tests are performed. Runtime analysis can, in similarity with DAST, only be performed when systems are in a functional and running state.

5.1.7.4 DAST + Runtime Analysis

Applications has to be functioning and running for DAST + runtime analysis to perform tests.

5.1.7.5 Summary

SAST does not require the system to be in a functional state to run tests and can therefore be integrated and start testing at the very beginning of software development processes. However, DAST, runtime analysis, as well as DAST + RA, needs a functioning and running system to conduct testing and can therefore be integrated later in development processes than SAST. Table 5.7 shows an estimated score for each technique.
5.1. EVALUATION OF AST IN CONTINUOUS INTEGRATION

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td>DAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST + RA</td>
</tr>
</tbody>
</table>

Table 5.7: Estimated score of the early testing property

5.1.8 Simple integration

More thoroughly explained rationale and criteria for simple integration is found under section 3.7.8 at page 25.

Simple integration is a property that relates to the agile mind-set and simplicity is stated in the agile manifesto.

(10) Simplicity—the art of maximizing the amount of work not done—is essential

Integration of tools should require a minimum amount of work; otherwise it does not fit into an agile process, and thus such testing will probably be disregarded.

It should be noted that the difficulty of integrating and installing tool depends on specific implementations. Only general concepts are discussed here.

5.1.8.1 SAST

SAST does not require any testing environments to be set up, and can therefore be judged as easy to integrate in CI systems. Some CI systems include plugins for static analysis tools, which makes the integration process even easier.

5.1.8.2 DAST

DAST requires a test environment where systems are always in functional and running states. Setting up such testing environments in a CI system is possible, but can be thought of as a more difficult process than integration of SAST, where systems do not even require compiling.

5.1.8.3 Runtime Analysis

As runtime analysis uses the instrumentation technology, it is integrated by including an executable file prior to the building process. This is a fairly easy process, as it does not require setting up an explicit testing environment.

5.1.8.4 DAST + Runtime Analysis

Integrating DAST + runtime analysis should be comparable with integration of both the techniques separately.

5\(^a\)RA do require a running test environment, but can use the same environment as used for functional testing.
5.1.8.5 Summary

SAST does not require any testing environment to be set up, and is also often supported as plugins to most CI tools. DAST, RA and DAST + RA need functioning test environments to be set up. However, if functional testing is performed within the project such environment already exists. Except from setting up the test environment, the difficulty of incorporating a security testing tool depends on specific tools. Table 5.8 shows an estimated score for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td>DAST</td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Estimated score of the *simple integration* property

5.1.9 Property evaluation summary

Table 5.9 shows a summary of all property evaluation.
5.2 Evaluation of Security Requirement Verification using AST

The evaluation does not consider different implementations of the techniques, but instead focus on how the techniques function and what scope they have.

5.2.1 A1 - Injection

*Injection* is further explained in section 2.5.1 at page 13.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

5.2.1.1 SAST
SAST applies taint analysis in order to detect injection vulnerabilities. Taint analysis work fairly well for general problems where applications use standard APIs, but runs in to problem when they don’t. Tools won’t for example recognize validation and escaping methods developed in-house, thus generating false positives. How this is dealt with depend on specific implementations of tools, which should be considered when choosing tools. SAST also has the limitation of only achieving a testing scope as wide as the source files it can access, thus having to make assumptions about unreachable third party libraries. Furthermore, taint analysis can only detect if validators are put in place and not if they have weaknesses in their implementation, which could be seen as a limitation.

5.2.1.2 DAST
Penetration testing detects injection vulnerabilities by locating input sources using spiders and analyzing them with vulnerability scanners and fuzzers. Such method can easily detect whether validation is put in place as well as trying to brake validations by trying a large amount of forged inputs. However, the drawback of dynamically finding injection vulnerabilities using black-box techniques is that the analysis has to deduce, only from responses of a system, whether an injection actually happened or not. This can be difficult as systems behave differently and therefore generate somewhat different responses.

5.2.1.3 Runtime Analysis
Runtime analysis utilizes similar techniques as SAST to detect injection flaws, with the difference of tagging real data in runtime instead of building estimated structures of control and data flow. This should be able to lower the amount of false positives of A1. Runtime analysis can also access third party libraries used in the same virtual machine, even if pre-compiled, and is thus able to construct a more comprehensive view of what’s happening. However, runtime analysis suffer from the same limitations as SAST by not being able to manage in-house developed APIs. Furthermore, RA does not assure full code coverage\(^6\) and therefore might fail to report some vulnerabilities.

5.2.1.4 DAST + Runtime Analysis
The mixture of DAST and runtime analysis should in theory be able to increase the testing scope, as well as reduce false positives by validating findings from two different techniques.

5.2.1.5 Summary
SAST and RA uses similar detection techniques, but with runtime analysis gaining the advantage of not needing to make any assumptions of how data flows through systems. DAST detects vulnerabilities by actually injecting data into systems, but struggles with non-generic responses and insufficient code coverage. However, by combining DAST and RA a broad test coverage as well as low level of false positives should be possible to achieve. Table 5.10 shows an estimated coverage for each technique.

\(^6\)As mentioned in 4.5.2 at page 45, the test coverage of RA depends on external testing methods executing applications.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RA</td>
</tr>
</tbody>
</table>

Table 5.10: Estimated coverage of A1 - Injection

5.2.2 A2 - Broken Authentication and Session Management

A2 - Broken Authentication and Session Management is further explained in section 2.5.2 at page 13.

5.2.2.1 SAST

Most of the vulnerabilities related to A2 are associated with session ID management. Some may be possible to detect by looking at usage of generic session management APIs, but most of them require knowledge about how session IDs are communicated between clients and servers to be detected and are therefore out of scope to be found by SAST. SAST, however, should be able to detect vulnerabilities such as passwords not being hashed before sent to databases or clients if able to recognize which data that are sensitive credentials.

5.2.2.2 DAST

As DAST can analyze actual session IDs sent back and forth to applications, it should be able to detect vulnerabilities such as URL exposure, session fixation, no session timeout, no session rotation\(^7\) and unencrypted sending of credentials. However, DAST cannot know about internal structures and thus fails to detect unencrypted storage of sensitive data.

5.2.2.3 Runtime Analysis

Runtime analysis should be able to perform a kind of combination of DAST and SAST analysis, but will not be able to test for vulnerabilities such as session ID timeouts and fixation attacks. Detecting such vulnerabilities requires actively forged requests, which cannot be performed by RA. However, vulnerabilities concerning proper session invalidation and rotation require manual configuration to be detected but that applies for the other testing techniques as well.

5.2.2.4 DAST + Runtime Analysis

The combination of DAST and RA probably increases the range of possible findings. DAST can help detecting vulnerabilities that require active testing, such as session fixation and rotation, while runtime analysis can detect vulnerabilities that require white-box knowledge, such as unencrypted storages.

\(^7\)Session rotation is the procedure of changing session IDs when users alternate between different authority-levels, for example login and logouts.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

5.2.2.5 Summary
At least in theory, the combination of runtime analysis and DAST should be able to detect most variants of vulnerabilities, but fails for those that require human expertise, such as invalidation and rotation of sessions. DAST and runtime analysis both cover some vulnerability types, while SAST only enables detection of vulnerabilities that only depend on the inner structures of applications, such as unencrypted storage of credentials. Table 5.11 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.11: Estimated coverage of A2 - Broken Authentication and Session Management

5.2.3 A3 - Cross-site Scripting (XSS)
A3 - Cross-site Scripting (XSS) is further explained in section 2.5.3 at page 14.

5.2.3.1 SAST
XSS is a subset of A1, and is detected in the same manner by applying taint analysis. The advantages and drawbacks also remains the same for these kinds of vulnerabilities.

5.2.3.2 DAST
The techniques for detection of XSS using DAST are also similar to those for A1. Scripts are injected as input sources and responses are analyzed. Reflected (or non-persistent) XSS is easier to detect than stored (or persistent), as the immediate response contains enough information to make judgments. Stored XSS is trickier detecting because scripts might have successfully been included into a database, but might be displayed later or at some other page.

5.2.3.3 Runtime Analysis
Applies the same techniques as for A1.

5.2.3.4 DAST + Runtime Analysis
With the help of runtime analysis, DAST should be able to easily detect stored XSS by gaining knowledge about how data flows internally.

5.2.3.5 Summary
As XSS is a subset of A1, the detection techniques and results are similar. Table 5.12 shows an estimated coverage for each technique.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

5.2.4 A4 - Insecure Direct Object Reference

A4 - Insecure Direct Object Reference is further explained in section 2.5.4 at page 14.

5.2.4.1 SAST

SAST should be able to detect whether objects are referenced without access control checks. However, such methods will result in a high amount of false positives as all object accesses don’t require access checks. Knowing which objects that actually does require access checks is very difficult for an automated analysis, because that most likely requires human expertise.

5.2.4.2 DAST

DAST can identify direct object references by modifying input data and analysing if new objects are possible to access. However, as for SAST, DAST cannot automatically determine whether such access is insecure or not.

5.2.4.3 Runtime Analysis

RA has the same problems with A4 as SAST.

5.2.4.4 DAST + Runtime Analysis

DAST + RA has the same problems with A4 as DAST and RA independently.

5.2.4.5 Summary

Table 5.13 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
<td>RA</td>
</tr>
</tbody>
</table>

Table 5.12: Estimated coverage of A3 - Cross-site Scripting (XSS)

Table 5.13: Estimated coverage of A4 - Insecure Direct Object Reference
5.2.5 A5 - Security Misconfiguration

A5 - Security Misconfiguration is further explained in section 2.5.5 at page 14.

5.2.5.1 SAST

SAST should have access to most configuration files within applications and can use static configuration analysis to detect security misconfigurations.

5.2.5.2 DAST

DAST does not have access to any configuration files but can detect some weaknesses by scanning configurations from the outside. Such scanning can for example detect usage of weak SSL or other web-server related issues.

5.2.5.3 Runtime Analysis

Uses the same methods and testing scope as SAST.

5.2.5.4 DAST + Runtime Analysis

Both has access to configuration files as well as being able to test SSL and web-configurations from the outside.

5.2.5.5 Summary

Table 5.14 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td>DAST</td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.14: Estimated coverage of A5 - Security Misconfiguration

5.2.6 A6 - Sensitive Data Exposure

A6 - Sensitive Data Exposure is further explained in section 2.5.6 at page 14.

5.2.6.1 SAST

Working from the inside, SAST can for example detect usage of known weak encryption APIs or if credentials are stored without security measures. However, SAST fails to detect actual exposures at it cannot track data outside applications. It’s often difficult to determine which data that is sensitive and should be secured, which is why A6 can be difficult to detect automatically.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

5.2.6.2 DAST

If DAST tools somehow can identify which data that is sensitive, DAST enables to see if the data is unencrypted, and therefore also exposed. However, obtaining such knowledge is difficult to automate.

5.2.6.3 Runtime Analysis

Runtime analysis suffers from similar problems as SAST when detecting A6.

5.2.6.4 DAST + Runtime Analysis

DAST + RA has similar issues as DAST and RA.

5.2.6.5 Summary

Table 5.15 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.15: Estimated coverage of A6 - Sensitive Data Exposure

5.2.7 A7 - Missing function level access control

A7 - Missing function level access control is further explained in section 2.5.7 at page 15.

A7 has similarities with A4 - Insecure Direct Object Reference, but concerns functions and methods instead of objects. As for A4, it is possible for automated tools to detect a possible lack of function level access control, but determining whether such access is allowed or not often requires human knowledge and expertise.

5.2.7.1 SAST

Detecting missing access control of some sensitive generic APIs might be possible, otherwise the same as for A4.

5.2.7.2 DAST

Requires human expertise. Same as for A4.

5.2.7.3 Runtime Analysis

Requires human expertise. Same as for A4.
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST

5.2.7.4 DAST + Runtime Analysis
Requires human expertise. Same as for A4.

5.2.7.5 Summary
None of the testing techniques supports good automated detection of A7, as it requires human knowledge to determine whether function access is allowed or not. Table 5.16 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.16: Estimated coverage of A7 - Missing function level access control

5.2.8 A8 - Cross-site Request Forgery (CSRF)

A8 - Cross-site Request Forgery (CSRF) is further explained in section 2.5.8 at page 15.

To prevent against CSRF, sites have to make sure that actual users perform each incoming request. Usually this is done by sending unpredictable tokens with each request to confirm that the request was not forged. Detection of CSRF vulnerabilities deals with finding if such tokens are used and how.

5.2.8.1 SAST
It should be possible to detect if CSRF tokens are used for incoming requests if applications use standard APIs to manage CSRF tokens. Doing so, SAST can detect whether all incoming requests contain CSRF tokens. If not, a vulnerability might have been found.

5.2.8.2 DAST
After spidering a site, DAST tools can check that all possible inputs contain CSRF tokens. It is also possible to actively test the randomness of tokens by assembling large amounts of tokens to perform calculations on.

5.2.8.3 Runtime Analysis
In theory, runtime analysis should be able to perform both methods used by SAST and DAST, as it both can access which APIs that are used as well as actual tokens used. However, detecting weaknesses in randomness requires large amounts of tokens, which can be difficult obtaining if not actively attempting to.
5.2.8.4 DAST + Runtime Analysis

The combination of DAST and runtime analysis should probably be able to detect most missing CSRF tokens as it can use techniques explained in both DAST and RA.

5.2.8.5 Summary

All techniques can probably detect most missing CSRF tokens for code that is analyzed. It should however be noted that the techniques used for detecting CSRF vulnerabilities might result in false positives. Table 5.17 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.17: Estimated coverage of A8 - Cross-site Request Forgery (CSRF)

5.2.9 A9 - Using components with known vulnerabilities

A9 - Using components with known vulnerabilities is further explained in section 2.5.9 at page 15.

5.2.9.1 SAST

SAST should have information about which components that is used within projects. Using blacklist techniques to compare with components that are known to contain vulnerabilities enables detection of vulnerable components. The blacklist has to be continuously kept up to date with new vulnerable components to be useful.

5.2.9.2 DAST

Dynamic scanners do not know which inner components that are used and can therefore not detect those with known vulnerabilities. However, they might be able to find vulnerabilities in third party components when testing systems, but such findings does not belong to A9.

5.2.9.3 Runtime Analysis

Same as for SAST.

5.2.9.4 DAST + Runtime Analysis

Cannot detect anything with the DAST component, but RA detects same vulnerabilities as SAST.
5.2.9.5 Summary

Table 5.18 shows an estimated coverage for each technique.

<table>
<thead>
<tr>
<th></th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAST</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAST + RA</td>
<td></td>
<td></td>
<td>DAST</td>
</tr>
</tbody>
</table>

Table 5.18: Estimated coverage of A9 - Using components with known vulnerabilities

5.2.10 A10 - Unvalidated Redirects and Forwards

A10 - Unvalidated Redirects and Forwards is further explained in section 2.5.10 at page 15.

5.2.10.1 SAST

SAST can locate calls to APIs for redirects and forwards and use taint analysis to detect if unvalidated user parameters are used when calculating destination addresses.

5.2.10.2 DAST

DAST can look for generated redirects when spidering applications and analyze the input parameters in order to determine whether the target address might be forgeable.

5.2.10.3 Runtime Analysis

Runtime analysis should be able to use similar techniques as SAST to detect A10 vulnerabilities.

5.2.10.4 DAST + Runtime Analysis

A combination of DAST and RA can apply both methods explained to perform analysis, which should be able to provide the most comprehensive results.

5.2.10.5 Summary

Table 5.19 shows an estimated coverage for each technique.
### Table 5.19: Estimated coverage of A10 - Unvalidated Redirects and Forwards

<table>
<thead>
<tr>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAST + RA</td>
<td>SAST</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RA</td>
</tr>
</tbody>
</table>

- **Good**: DAST + RA
- **OK**: SAST, DAST
- **Bad**: RA
5.2. EVALUATION OF SECURITY REQUIREMENT VERIFICATION USING AST
### 5.2.11 Summary of evaluation of Security Requirement Verification

<table>
<thead>
<tr>
<th>Property</th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 - Injection</td>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2 - Broken Authentication and Session Management</td>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A3 - Cross-site Scripting (XSS)</td>
<td>DAST + RA</td>
<td>DAST</td>
<td>SAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4 - Insecure Direct Object Reference</td>
<td></td>
<td>SAST</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A5 - Security Misconfiguration</td>
<td>DAST + RA</td>
<td>DAST</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SAST</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6 - Sensitive Data Exposure</td>
<td></td>
<td>SAST</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A7 - Missing function level access control</td>
<td></td>
<td>SAST</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A8 - Cross-site Request Forgery (CSRF)</td>
<td>SAST</td>
<td>DAST</td>
<td>RA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DAST + RA</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.20: Estimated scores of all risks in OWASP Top Ten
5.2. Evaluation of Security Requirement Verification Using AST

<table>
<thead>
<tr>
<th>Property</th>
<th>Good</th>
<th>OK</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>A9 - Using components with known vulnerabilities</td>
<td>SAST</td>
<td>RA</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td>RA</td>
<td>DAST + RA</td>
<td></td>
</tr>
<tr>
<td>A10 - Unvalidated Redirects and Forwards</td>
<td>DAST + RA</td>
<td>SAST</td>
<td>DAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RA</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.21: Continuing: Estimated scores of all risks in OWASP Top Ten
Chapter 6

Conclusion

This chapter discusses the result of the analysis. It also addresses the applicability of the methodology, suggests potential future work, and defines its contributions to the field.

Agile development methodologies have become increasingly popular, especially in projects building web applications. However, incorporation of software security in lightweight approaches can for various reasons be difficult. Using automated security testing techniques to detect vulnerabilities throughout the complete development processes is one method that strives to build security into agile projects. This thesis studies existing security testing techniques and evaluates both how they technically fit into a continuous integration environment and how they adhere to agile principles as well as what level of security they can assure. A method was formed that intended to evaluate how well different security testing techniques fit into continuous integration environments and which security requirements they could verify. The method and results from the analysis is further discussed here.

6.1 Discussion

The purpose of this thesis was to investigate which, and to what extent, security requirements are possible to verify in continuous integration processes for web applications, using existing security testing techniques. This was realized by identifying, studying and categorizing existing security testing techniques and forming a method to analyze how well they fit into CI environments as well as comparing which security risks\(^1\) they can help avoiding. As the purpose indicates, the goal was not to find a final solution to all security problems in agile development, but rather to investigate the field of integrating automated security testing into CI, and to evaluate its possibilities and limitations. The final work could be used as a source of information and as a pinpointer to important aspects that should be considered when incorporating security testing into CI environments, rather than providing final judgement of which tools to use.

The first section of the analysis indicates that there is no security testing technique that could be thought of as a perfect fit and an obvious choice for usage in continuous integration environments. Which technique to use depends on how specific projects are formed and

---

\(^1\)It was established in chapter 2 that security requirements are usually based on security risks, as security requirements are commonly non-functional and therefore difficult proving.
which of the properties that are considered important. SAST, for example, adheres to the simplicity of agile, as it is easy to integrate and provides the possibility of early testing with a large test coverage. However, its testing results can contain a large amount of false positives and false negatives, which requires both security expertise and manual work to resolve. DAST has a lower level of false positives and false negatives in its results, but is not as easy to incorporate into a CI environment as SAST. It requires systems to function as well as a running testing environment to enable testing. It also obtains a more narrow and unknown test coverage than other techniques. Runtime analysis is a relatively new security testing method that seems to be easy to integrate in a CI environment, but instead requires the project to perform additional testing in order to passively analyze systems. If such functional testing is utilized, runtime analysis might be a good choice for usage in CI. The combination of DAST + RA resolves most of the drawbacks of DAST and RA when used independently, but might still be difficult to incorporate in a CI environment as it, in similarity with DAST, requires a running test environment.

Security risks and vulnerabilities is a field that continuously evolves, as hackers constantly find new ways of exploiting web applications. The analysis technique used for this thesis therefore focus only on the most common and dangerous security risks as of now. The results indicate that no technique is comprehensive in its findings, as automated security testing struggles to detect vulnerabilities where it’s required to have knowledge of what’s allowed in a specific system. The fact that no technique can uncover all vulnerabilities automatically has to be acknowledged, as manual testing should always be considered to some extent when using automated security testing techniques. The analysis, however, indicates that which kinds of vulnerabilities that the different security testing techniques enables detecting mainly depend on which structures the tools has knowledge about; inner, external, or both. SAST, for example, use white-box testing techniques, which both supports detection of some technical vulnerabilities such as injections and XSS, as well as vulnerabilities related to misconfigurations and usage of known vulnerable components. Detecting such vulnerabilities requires access to configuration files as well as knowledge of which components that are used, which in most cases is accessible when using white-box testing. Dynamic analysis, DAST, cannot detect such vulnerabilities as it performs testing without information about applications internal structures. The analysis further shows that DAST seldom performs better than SAST for detection of vulnerabilities. However, it should be noted that SAST usually results in more false positives than DAST, making them difficult to compare. Runtime Analysis performs testing with knowledge of both inner and external structures of applications, so called grey-box testing, but still has a very similar score to SAST. This mainly depends on the fact that testing from the outside does not improve the range of possible findings in a distinguishable way. SAST and RA both utilize similar methods to detect vulnerabilities, with the difference that SAST performs control and data flow analysis to track the flow of data while RA uses actual data in runtime as applications execute. As RA uses real data, it should be able to result in less false positives than SAST. The combination of DAST and RA supported the widest possible findings of vulnerabilities in the analysis. This is because it enables verification of each finding from two completely different testing techniques, which should be able to lower the rate of false positives drastically. The combination also enables DAST to be fed with information of what to test from the Runtime agent, which can improve the poor test coverage of DAST tools, which otherwise is a main drawback of dynamic testing. Using the combination of DAST and RA however is not trivial, and is today mainly used by leading vendors in the field of security testing.
6.2. CONTRIBUTIONS

The method used for this thesis is qualitative and theoretical, which can limit its usefulness. The qualitative analysis deduces the capabilities of each security testing technique by how it theoretically performs testing, instead of evaluating actual tools. The scores presented in the analysis section are based only on knowledge and information presented in the theoretical section of the report, making them sensitive to information that is wrong or misinterpreted. A possible extension of the method could be to also include a quantitative and practical study of different techniques and specific implementations. Carrying out such an evaluation in a fair manner was considered too difficult and time-consuming for the scope of this thesis. However, the qualitative analysis can provide general guidelines of what to consider when deploying security testing within continuous integration environments and also pinpoints obvious advantages and drawbacks of the distinct techniques. Further, one important aspect to consider when using qualitative analysis methods is that the results are sensitive to inaccurate or erroneous facts. As mentioned in the evaluation of sources, the section of the report discussing IAST is supported with less academic references than the other security testing techniques. How this might have affected the outcome of this thesis is obviously difficult to know, but it should be observed and regarded when reading the report. It should also be noted that the method was assembled only for the purpose of this thesis and has not been tested or criticized externally. The method utilizes non-exhaustive lists of risks and properties that might limit possible conclusions. For example, the method applies OWASP Top Ten in order to compare the possible findings of vulnerabilities for each security testing technique. OWASP Top Ten, however, only concerns the most important and dangerous security risks according to OWASP. How well the testing techniques function for other risks than those in OWASP Top Ten is outside of this thesis’ scope. Furthermore, the list of CI properties used in the method was constructed only for the purpose of this thesis, as no usable list to compare techniques within CI environments were identified. This list adheres to technical properties of CI as well as basic agile principles. The method can be re-used to evaluate additional security testing techniques, as the market of tools and techniques evolves rapidly. Such evaluation requires a qualitative study and analysis of the testing techniques. Finally, the qualitative method did help to perform a high-level study of the field of security testing techniques in continuous integration environments. The result is presented as a list of identified possibilities and drawbacks of each method, which can be used as a general guideline when implementing security testing in CI.

6.2 Contributions

During the research phase of this thesis, no study containing a complete compilation of existing security testing techniques were identified. This thesis attempts to contribute with a somewhat complete list of categorized techniques that explains fundamental differences. Furthermore, no independent research of IAST was found when writing this thesis, which to a limited extent is attempted in this thesis. The thesis also attempts to describe the complicated nature of security requirements and why they often are managed using security risks. It also describes how security risks relate to security testing techniques. Finally, the thesis is built upon a method that evaluates how well different security testing techniques fit into CI environments as well as analysing how well they test against OWASP Top Ten. This evaluation can hopefully be used as a guide when deciding which security
testing technique to use in continuous integration environments, as well as helping to determine the limitations of such methods. As mentioned earlier, the method itself could also be re-used if new techniques that need evaluation are identified.

6.3 Recommendations and Future work

The scope of this thesis had to be limited to a theoretical and qualitative study of security testing techniques. However, a practical comparison of the techniques is preferred to perform a more thorough evaluation. For such a comparison to be trustworthy, it should include several tools from each security testing technique and evaluate how they perform in a real development process using continuous integration.

As no independent study of IAST was found during the writing of this thesis, it’s a field that definitely requires future study.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Pontus Thulin]