



DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

Integrating Automated Security Testing in the Agile Development Process

EARLIER VULNERABILITY DETECTION IN AN
ENVIRONMENT WITH HIGH SECURITY
DEMANDS

ANDREAS BROSTRÖM

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

Integrating Automated Security Testing in the Agile Development Process

Earlier Vulnerability Detection in an Environment with High Security Demands

Integrering av automatiserad säkerhetstestning i den agila utvecklingsprocessen

Upptäck sårbarheter tidigare i en miljö med höga säkerhetskrav

ANDREAS BROSTRÖM
<abros@kth.se>

DA225X, Master's Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Royal Institute of Technology year 2015

Supervisor at CSC was Linda Kann
Examiner was Mads Dam
Employer was Nordnet Bank AB
Supervisor at Nordnet was Joakim Hollstrand

June 22, 2015

Abstract

The number of vulnerabilities discovered in software has been growing fast the last few years. At the same time the Agile method has quickly become one of the most popular methods for software development. However, it contains no mention of security, and since security is not traditionally agile it is hard to develop secure software using the Agile method. To make software secure, security testing must be included in the development process.

The aim of this thesis is to investigate how and where security can be *integrated* in the Agile development process when developing web applications. In the thesis some possible approaches for this are presented, one of which is to use a web application security scanner. The crawling and detection abilities of four scanners are compared, on scanner evaluation applications and on applications made by Nordnet. An example implementation of one of those scanners is made in the testing phase of the development process. Finally, a guide is created that explains how to use the implementation.

I reach the conclusion that it is possible to integrate security in the Agile development process by using a web application security scanner during testing. This approach requires a minimal effort to set up, is highly automated and it makes the Agile development process secure and more effective.

Referat

Integrering av automatiserad säkerhetstestning i den agila utvecklingsprocessen

Antalet upptäckta sårbarheter i mjukvara har ökat fort under de senaste åren. Den agila metoden har samtidigt blivit en av de mest populära metoderna för mjukvaruutveckling. Den berör dock inte säkerhet, och eftersom säkerhet, traditionellt sett, inte är agilt så blir det svårt att utveckla säkra mjukvara med den agila metoden. För att kunna göra mjukvaran säker så måste säkerhetstestning infogas i utvecklingsfasen.

Syftet med det här arbetet är att undersöka hur och var säkerhet kan *integreras* i den agila utvecklingsprocessen vid utveckling av webbapplikationer. Några sätt att göra detta på beskrivs i arbetet, varav ett är att använda ett verktyg för säkerhetstestning. En jämförelse av hur bra fyra sådana verktyg är på att genomsöka och hitta sårbarheter utförs på applikationer designade för att utvärdera verktyg för säkerhetstestning, samt hos Nordnets egna applikationer. Sedan beskrivs en exempelimplementation av ett av dessa verktyg i testfasen av utvecklingsprocessen. Slutligen, tas en guide fram som beskriver hur implementationen kan användas.

Jag kommer fram till att det är möjligt att inkludera säkerhet i den agila utvecklingsprocessen genom att använda ett verktyg för säkerhetstestning i testfasen av utvecklingsprocessen. Detta tillvägagångssätt innebär en minimal ansträngning att införa, är automatiserat i hög grad och det gör den agila utvecklingsprocessen säker och mer effektiv.

Acknowledgments

This thesis describes my degree project conducted for the School of Computer Science and Communication (CSC) at KTH Royal Institute of Technology. I would like to thank everyone that has helped or contributed in any way and supported me and my work. This has been a great experience, and I am very pleased with the results.

I want to direct a special thank you to the following people:

Linda Kann and **Mads Dam**, *KTH*, for their advice, support and feedback, and for taking me on as a thesis student.

Joakim Hollstrand, *Nordnet*, for his advice, assistance, support and feedback, and for taking me on as a thesis student.

Finn Hermansson, *Nordnet*, for his feedback, and for giving me the opportunity to do this.

Tasos Laskos, *Arachni*, for his assistance and advice with the final implementation.

Maja Rinnert, for all her love and support.

Contents

- Glossary** **1**

- 1 Introduction** **3**
 - 1.1 Preface 3
 - 1.2 Problem statement 3
 - 1.3 Purpose 4
 - 1.4 Outline 4

- 2 Background** **7**
 - 2.1 About Nordnet 7
 - 2.2 Agile software development 8
 - 2.2.1 Continuous integration 10
 - 2.3 Software testing 11
 - 2.3.1 Approaches 12
 - 2.3.2 Automation 13
 - 2.4 Web application security 13
 - 2.4.1 Common web application vulnerabilities 15
 - 2.4.2 Security testing approaches 18
 - 2.5 Agile security testing 19
 - 2.5.1 Source code analysis tools 21
 - 2.5.2 Vulnerability scanners 22

- 3 Approach** **25**
 - 3.1 Pre study 25
 - 3.1.1 Evaluating approaches for agile security testing 25
 - 3.1.2 Interview study 26
 - 3.2 Evaluating scanners 27
 - 3.2.1 Gathering requirements 28
 - 3.2.2 Comparing scanners 28
 - 3.3 Testing 29
 - 3.3.1 Environment 29
 - 3.3.2 Execution 31
 - 3.4 Integration and Implementation 33

3.4.1	Integration	33
3.4.2	Implementation	34
3.5	Guide	35
4	Results	37
4.1	Pre study	37
4.2	Evaluating scanners	38
4.2.1	Gathering requirements	39
4.2.2	Comparing scanners	40
4.3	Testing	45
4.3.1	WIVET	45
4.3.2	WAVSEP	48
4.3.3	Nordnet	56
4.4	Integration and implementation	56
4.4.1	Integration	57
4.4.2	Implementation	59
4.5	Guide	60
5	Conclusions	65
5.1	Discussion	65
5.1.1	Goals	65
5.1.2	Results	67
5.2	Integrating Automated Security Testing in the Agile Development Process	70
5.3	Recommendations	71
5.4	Future work	72
	Bibliography	73
A	WAVSEP results	79
A.1	Arachni	79
A.2	Burp Suite	82
A.3	OWASP ZAP	85
A.4	w3af	88
B	Guide	91

Glossary

attack

A deliberate attempt to assault system security, evade security services or violate the security policy of a system.

attack vector

A route or method used to get into computer systems by taking advantage of known weak spots.

attacker

Someone, or something, that attacks a system by using different attack vectors.

build

A build refers to a row in a delivery pipeline. If one of the steps ends with a failure, the whole build is marked as a failure.

CD

Continuous Delivery is an extension of CI below. The goal of this extension is to ensure that the software can be released to production at any time.

CI

Continuous Integration is a software engineering practice with the goal that all members of a team should integrate their work frequently.

delivery pipeline

A step-divided pipeline where each step is an activity in the CD (or CI), from commit to finished product ready to be deployed. Each step in the pipeline must succeed in order to reach the next step, and eventually the end.

exploit

An attack on a system that takes advantage of a particular vulnerability found on the target system.

penetration testing

Performing an attack on a computer system with the intention of uncovering potential vulnerabilities.

QA

Quality Assurance is the process of checking if a product under development is meeting specified requirements. It is used to avoid and prevent mistakes or defects in these products.

scanner

Short for vulnerability scanner, which is a program that scans a given system to find any vulnerabilities, often used in penetration testing.

SDLC

System Development Life Cycle refers to the process of developing (planning, creating, testing and deploying) a system of software, hardware, or a combination of both.

vulnerability

A security weakness or risk in a system that could allow an attacker to attack the system and thereby compromise its integrity, availability, or confidentiality.

XSS

An abbreviation of Cross (X) Site (S) Scripting (S), which is a common attack vector on web applications.

Chapter 1

Introduction

This chapter introduces the motivation behind this thesis, the problem statement, the purpose and an outline of the contents.

1.1 Preface

This degree project aims to investigate how security can be integrated in the Agile development process. The degree project is carried out at Nordnet Bank AB, which is a small Nordic bank. Their main area of business is investments and savings, but they also offer services in pension and banking [41]. They are fully web-based, and therefore they have high web application security demands.

1.2 Problem statement

The use of an Agile [7] work flow has become more common within all kinds of software development and there are many who claim that it works *really* well [32, 18, 26]. One of the features that has made agile development so successful is continuous integration (CI) [24], which makes it possible to automate integration and thereby further speeding up the process from idea to production ready code [24]. One major flaw with the Agile development process is that it does not say anything on how to achieve security; security is not even mentioned in the Agile Manifesto [7].

Today there are many companies that, like Nordnet, have adopted Agile, and where security is highly prioritized. This way of working leads to security being separated and disconnected from the development process into something that is not Agile. Since security still has to interact with the development process this has led to sort of a middle ground between the two processes; A development process that is neither truly Agile nor fully secure [11]. To achieve a development process that is both effective and secure a way of integrating security in Agile [55] is needed. There are a couple of proposals on how that can be achieved [55], but it seems like none of them can solve all problems, and either the proposals have not been implemented

(only theoretical solutions) or they have not proven to be as useful as suggested, which has led to them not being used to a greater extent [55].

I have therefore chosen to investigate how security can be integrated in the Agile development process, and more specifically; *how can automated security tests be included in the Agile development process in order to avoid vulnerabilities in production?*

1.3 Purpose

This thesis therefore intends to find a way to integrate security in the Agile development process. Since Nordnet is working with CI and they are mainly developing web applications, this will also be the main focus of this thesis, but the results should be applicable to any Agile development process.

More specifically, the following things are addressed in the thesis:

- An investigation of how security can be integrated in the Agile development process.
- An investigation of how and where automated security tests can be included in CI within the Agile development process.
- Producing a guide for developers, telling them how to set up automated security tests, and helping them to understand security implications and risks and showing them how they can mitigate certain risks to reduce vulnerabilities.

In order to:

- Achieve a higher security awareness among developers.
- Create an easy and effective way for developers to verify that their code is secure.

To my knowledge, no comparable efforts have been made to secure the Agile development process by using an automated web application security scanner that is seamlessly integrated in the delivery pipeline.

1.4 Outline

This thesis starts with an abstract that summarizes this thesis. After that there is a small section where I take up some notable acknowledgements. A table of contents follows to give an overview of the document. The rest of thesis consists of:

- A terminology section that includes abbreviations and terms regularly used in the thesis.

1.4. OUTLINE

- The introduction in [chapter 1](#). This is meant to introduce the reader to the problem and purpose of this thesis, and present what the rest of the thesis will cover.
- The background in [chapter 2](#). It contains in depth theory needed to understand the rest of the thesis.
- The approach is presented in [chapter 3](#). It contains information about the methods used when attempting to solve the problem posed in the introduction.
- The results are presented in [chapter 4](#). It contains all results from the approach chapter.
- The conclusions of the work are presented in [chapter 5](#). It contains a discussion about the purpose and the results of this thesis.
- The last section contains the references.
- After that comes the appendices. They include some of the results in a more detailed form (see [Appendix A](#)) and a guide that was produced during this thesis (see [Appendix B](#)).

Chapter 2

Background

This chapter reviews the theory needed to understand the problem statement of the thesis and [chapter 3](#). This includes information about: Nordnet, Agile software development, software testing, web application security and agile security testing.

2.1 About Nordnet

Nordnet is a small bank in the Nordic region (Sweden, Denmark, Finland and Norway) with its main office located in Sweden, Alvik [41]. Nordnet was founded in 1996 and was then one of the first brokers in the Nordic region to offer online trading. Since then Nordnet has grown to almost 400 employees. Their main area of business is investments and savings, but they also offer services in pension and banking. Today it is first and foremost an IT company in the financial industry rather than a bank with an online presence. Security is always a high priority in a bank and in this case a big part of that is IT security [40], and more specifically web application security.

Banking is built around trust. Customers that deposit money to a bank trust that they can later withdraw the same amount, they trust that nothing *bad* will happen to their valuables while they are stored at the bank; they trust that their bank is secure. Therefore, any security related incidents will generate a great deal of badwill towards the bank, which means that they will lose customers. Banks are also under strict regulations from inspection bodies (such as Finansinspektionen in Sweden [22]) and any security related incidents could therefore also lead to additional penalties, like fines. One could argue that security is more important to Nordnet than to bigger banks, since losing customers and incurring fines will affect them more. Since Nordnet operates in all four Nordic countries they need to follow the regulations in all of them, which also gives them a reason to be extra cautious with security.

Nordnet has several processes for handling security, such as risk analysis and security testing through audits and penetration testing. Security testing is performed by both an internal security team on Nordnet's test environment, and by an external

company on both Nordnet’s test environment and production environment. All of these tests and audits occur frequently, but they rely on manual work and are not well integrated in the development process, which is investigated in this thesis.

2.2 Agile software development

There are many different methods used to develop software. The focus of this thesis is the *Agile* method, but in order to fully understand Agile it is first necessary to look at what existed before Agile; the more *traditional* methods. The most widespread traditional software development method that existed before Agile, and still does today, is probably the *Waterfall* method [54].

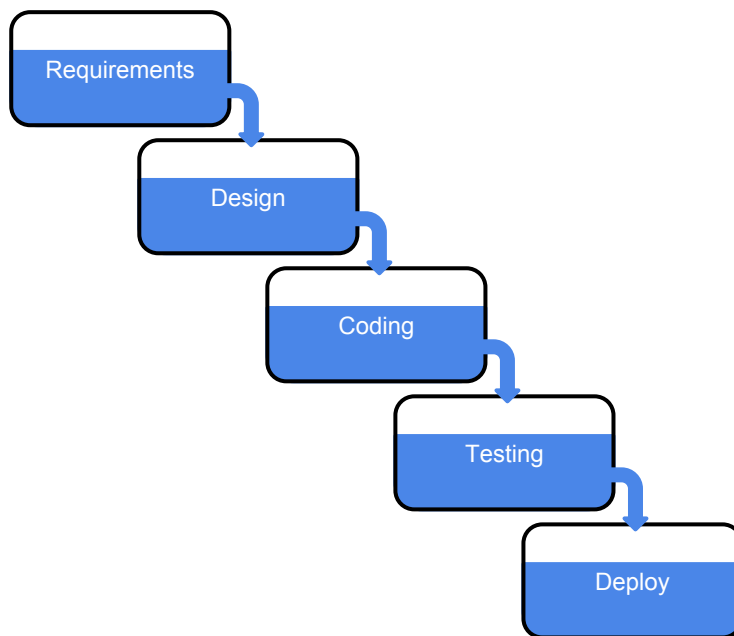


Figure 2.1. An overview of the SDLC according to the Waterfall method [54]

The system development life cycle (SDLC) according to the Waterfall method can be seen in [Figure 2.1](#). The core idea is that in order to develop good software it is necessary to have built a good foundation for it, which in this case mean requirements, documentation, design guidelines etc. The Waterfall method is highly sequential, which means that one phase must be fully completed in order to move on to the next. It also means that if something goes wrong in the *Testing* phase that has to do with an unclear requirement you must start over from the *Requirements* phase and go through all the phases again.

To cope with the problems of traditional methods, like the one mentioned above, new methodologies appeared in the late 1990’s [2]. In 2001 a group of people that

2.2. AGILE SOFTWARE DEVELOPMENT

worked with some of these new methodologies met in order to see if they could find a common ground. This led to the creation of The Agile Manifesto [7]. In short, Agile software development is a group of software development methods that aim to be lighter, faster and more efficient than traditional software development methods [4]. The manifesto recaps the ideology below [7]:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

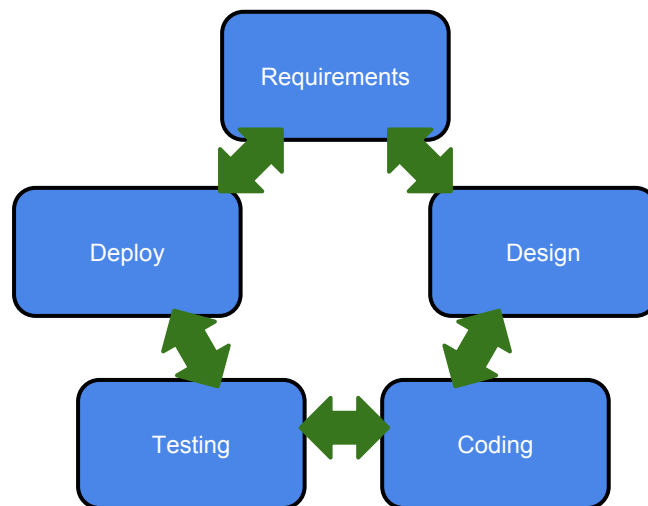


Figure 2.2. An overview of the SDLC according to the Agile method [2]

The SDLC according to Agile can be seen in [Figure 2.2](#). The core idea is to reach the delivery of software faster and then iteratively improving it by adding features and tests [7]. This means that the time to market with the Agile method is much faster than in the case with Waterfall, which can explain the numbers in [Figure 2.3](#), and the large adoption of Agile that we see today [18]. Requirements in the Agile method are handled via user stories, which each contain a high-level definition of a requirement containing enough information so that a developer can produce a reasonable estimate for how much effort is required to implement it.

Another reason for the success of Agile software development is the ability to start integrating a new system with existing ones from the start instead of waiting until the new project is done, this is called *continuous integration* [24].

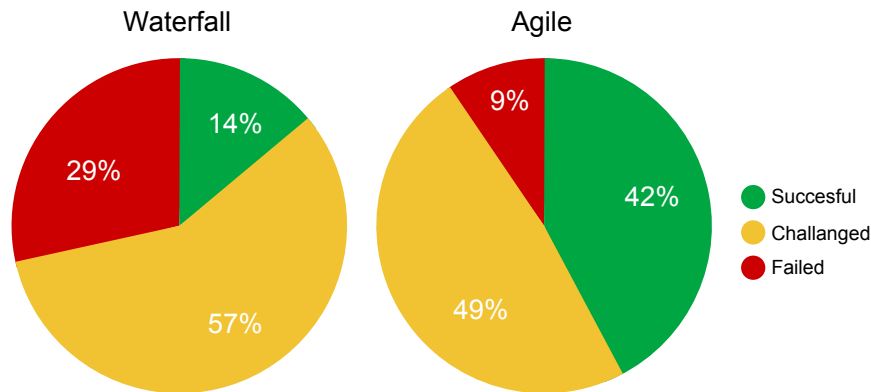


Figure 2.3. A comparison of the outcome for projects using the Waterfall or the Agile method in the CHAOS project database from 2002 to 2010 [58]

2.2.1 Continuous integration

Continuous integration (CI) is the practice of integrating code frequently during development. It was first introduced by Gary Booch in 1991 [10] and later redefined by Martin Fowler [24] to what it is today. CI aims to solve a problem that traditional development methods have, where integration is a long and unpredictable process. The core concepts of CI are automating the building, testing and deployment of software. [24]

In CI every change to software (e.g. by a commit in a source code management system) triggers a new build, which means that every change is not only built as a standalone product, but also integrated, tested and verified. CI allow teams to build, test and release their software on demand and if the cycle from change to deploy is fast the teams can get fast feedback and therefore quickly correct anything that might be wrong. This also means that there is always a way to quickly release a fully tested version to production if it is needed.

CI has grown since its introduction, and today there are many companies that offer tools that make CI easy to set up and use. One of the most popular tools is Jenkins CI [31], which also happens to be the tool that Nordnet use.

Figure 2.4 shows how a typical project on Nordnet is configured in Jenkins. It has a couple of steps, or phases, which must all be passed in order to complete a build. Brief explanations of the different steps follow:

- **Build and Package** - This step will fetch the latest code (that triggered the change), build the project, run all unit tests, and then package it.
- **CI: Automatic tests** - This step will deploy the project to an environment called CI, e.g. if the project is a tomcat web application it will be deployed

2.3. SOFTWARE TESTING

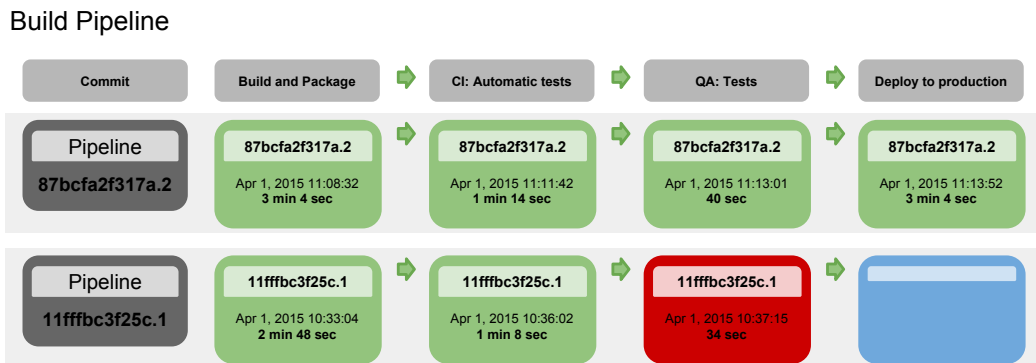


Figure 2.4. A generalized overview of the delivery pipeline used for a project at Nordnet. For more information about the layout of the image, see Jenkins CI [31]

to a tomcat server in that environment. This step also runs some integration tests on the code deployed in the CI environment.

- **QA: Tests** - This step is much like the above, with the difference that the project is deployed in another environment called TEST. It also includes automated and/or manual QA testing on the code deployed in the TEST environment.
- **Deploy to production** - This step deploys the project to the production environment.

Nordnet uses an approach called continuous delivery (CD), which is a natural extension of CI. When using CD all but the last of the steps described above are automated, i.e. when *Build and Package* is completed it will trigger the next step (*CI: Automatic tests*). When the *QA: Tests* step is complete, it will activate a button that has to be manually pressed in order to continue the CD process and start the *Deploy to production* step.

2.3 Software testing

The term bug was introduced in 1947 when a computer at Harvard University crashed due to a moth that had flown into the machine and got stuck between a set of relays. One of the goals of software testing is to find bugs. [49, Chapter 1]

The cost of fixing a bug is highly related to where in the process the bug is found as can be seen in [Figure 2.5](#). For example a particular bug might cost \$100 to fix if it is found in the testing phase but can cost millions of dollars if it is found when the software has already been shipped to the customer. Two other goals of software testing are therefore to find the bugs as early in the development process as possible and to make sure that they get fixed. [49, Chapter 1]

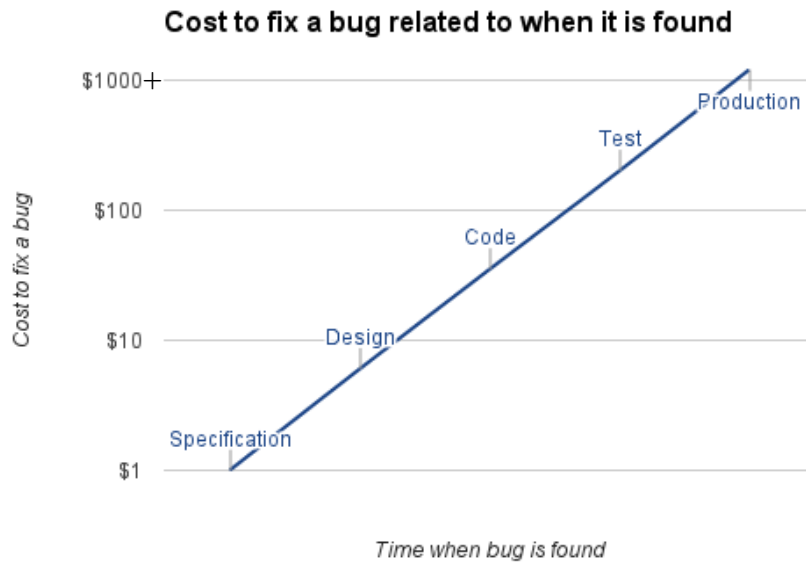


Figure 2.5. The cost of fixing bugs related to where it is found. This figure is an adapted version of the original from *Building Security In* [36]

Today there are many different approaches on how to perform software testing, there are even development methodologies built around testing, like test driven development [6]. In the following subsections different approaches to software testing are presented, and a section about testing automation, but first some information about things to watch out for.

Two of the biggest issues that can occur while testing are false positives and false negatives. In software testing a false positive is when normal and expected behavior is incorrectly identified as faulty, this means that the test case will fail when there is no real error. This is an issue since if many errors are incorrectly reported they can hide the real errors [48]. A false negative is related, but almost like the opposite, of a false positive. It is when an error is not found or not reported by the tests when it should have been, which can give a false sense of security [47].

2.3.1 Approaches

There are two approaches to software testing that are widely used [49, Chapter 4]:

- Static and dynamic testing
- White-box and black-box testing

Static testing is defined as examining or reviewing the code of the software to test, trying to identify bad practices to verify that it is written correctly. Dynamic

2.4. WEB APPLICATION SECURITY

testing is defined as testing the software by using and running it, trying to validate that it is correct. [49, Chapter 4]

White-box testing is related to static testing since it also targets the code of the software. In white-box testing the tester typically executes specific parts of the code to see if they work correctly. Black-box testing is similarly related to dynamic testing as it tests the software when it is running. In black-box testing the tester treats the software as a black box, she knows what it is supposed to do but she cannot look at exactly how it operates. [49, Chapter 4]

2.3.2 Automation

When testing a newer version of a software, everything that was tested in the previous versions must be tested again. This is not only time-consuming but also tiring and error-prone. To solve this issue tools used for testing were introduced as a way to code test cases that could be carried out by a computer, automated. [49, Chapter 15]

When the Agile software development methods became more popular the speed of testing became more important and automated testing became popular (see [section 2.2](#)). Some automated tests that are very popular today are:

- **Regression tests** - Tests that are introduced when bugs are fixed to make sure that they will not reoccur. [34, Chapter 9]
- **Unit tests** - A test that is meant to test a small unit of code to ensure that the particular unit has the desired function. A unit could for example be a method, a class or a file. [34, Chapter 10]
- **Integration tests** - Tests that test a combination of two or more software components, or units, together to find out if they work as intended. This could mean testing anything from the integration between two methods to testing the integration between the software and the production environment. [34, Chapter 11]

2.4 Web application security

The web is growing, and many companies are moving more of their business to the web. This is especially visible in the world of banking, where offices are closing down and the amount of cash management is declining. Instead, customer services have moved into interactive chat rooms, support email accounts and telephones, and cash management is no longer necessary when we have online banking and credit cards. As companies continue to increase their online footprint they also become a more attractive target for attackers. All new software developed by Nordnet today is either web applications, web-services or part web applications (hybrid applications), although they still maintain some legacy applications. The biggest security concern for Nordnet is to protect their customers, which means that they need to secure all

applications that their customers can interact with. Therefore, this section and this thesis will focus on the security of web applications.

Traditionally, the biggest issue when it comes to security seems to be that the presence of security has been seen as a feature, something that was nice to have, rather than regarding the absence of security as a bug [36]. This may seem strange now, but it makes more sense when considering that when the Internet was born, there was no such things as viruses or worms [57]. To defend against something that you have no idea exists is one of the dilemmas of security, another is trying to find and fix all possible vulnerabilities before an attacker can find and exploit them.

The attitude towards software security has shifted lately, mostly due to the big number of discovered vulnerabilities (as can be seen in Figure 2.6) and the literature that was written in response to that discovery [36]. As the IT sector keeps getting bigger and the software is getting more and more complex, the issue of security is also growing.

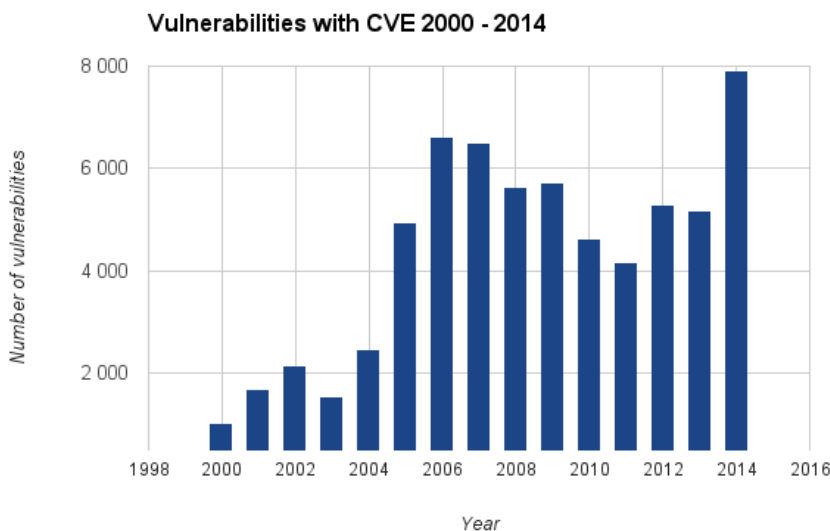


Figure 2.6. The number of vulnerabilities per year from January 2000 until December 2014 found in the National Vulnerability Database [39]

To try to solve this problem, Gary McGraw suggested an approach called the *Three pillars of security* [36]:

- **Applied Risk Management** - The process of trying to identify, rank and track risks. To classify the severity of vulnerabilities a system called CVSS [38] is used.
- **Software Security Touchpoints** - A set of best practices of how to integrate security in the SDLC, and what can be done in each phase of development.

2.4. WEB APPLICATION SECURITY

- **Knowledge** - The process of gathering, structuring, and sharing security knowledge to help provide a solid foundation for security best practices.

Web application security includes security of websites, web applications and web services. The majority of vulnerabilities in web applications come from bad coding practices, code complexity, flawed coding (bugs) and lack of testing [36, Chapter 1].

2.4.1 Common web application vulnerabilities

To keep track of vulnerabilities and other information in web application security there is an organization called the *Open Web Application Security Project* (OWASP) [46]. OWASP is a “worldwide not-for-profit organization focused on improving the security of software” [46]. Since it started in 2001 it has produced guides on how to approach development [43] and testing [44] with security in mind, kept track of the major threats to web applications and provide tips and tricks on how to mitigate and get rid of the vulnerabilities.

The OWASP top 10 project [45] lists the ten most critical vulnerabilities found in web applications. This list is based on statistics gathered by several organizations and companies [63, 23]. In the book *Introduction to Computer Networks and Cybersecurity* [65] the project is described like this:

“The OWASP Top Ten represents a broad consensus in identifying the most critical web application security flaws. An organization must begin by ensuring that their web applications do not contain these flaws, and adopting the OWASP Top Ten is the first step toward changing the software development into one that produces secure code.” [65, Chapter 26]

Web applications can contain both flaws found in regular applications, such as access control or authorization flaws, and some specific to web-based applications, such as cross-site scripting (XSS). The ten major threats against web applications identified in 2013 by OWASP [45, 37] are presented in a list below, and then their relevance will be discussed from the perspective of Nordnet. Specifically, the list will contain descriptions of the vulnerabilities, information on how the vulnerabilities manifest themselves and how they can be tested.

- **Injection** - The most common types of injection today are SQL injection and OS command injection. An injection occurs when untrusted data from a user is interpreted by a server as a command or a query. This type of vulnerability enables a user to execute arbitrary code on the server without any authentication. These flaws are generally easy to discover when examining code, and harder to discover with automated testing. However, they can be tested by locating places where user submitted data is injected into SQL queries or in OS commands, and trying to inject untrusted data to those places and see what happens. An example of this is to include double or single quotes in data that is later used to construct an SQL query, in hope of breaking out of the query. If this test leads to any form of exception or error it usually means that a vulnerability has been found.

- **Broken Authentication and Session Management** - When authentication or session management is implemented poorly, allowing an attacker to pose as valid user by compromising passwords, session tokens or other secrets. Some examples are: storing passwords in clear text, allowing easily guessable passwords, storing the session ID in the URL or sending any form of credentials over unencrypted connections. As these flaws are highly dependent on the implementation, they are often hard to find. A session ID stored in the URL is an example of something that is easy to test both manually and with an automated approach.
- **Cross-Site Scripting (XSS)** - Allows an attacker to execute malicious JavaScript code on a user's client. This happens when the user's client displays a value that is not properly escaped. This can be used to steal sensitive data such as credit card numbers, session cookies and social security numbers. XSS flaws are normally divided in two categories depending on where the untrusted data is used; on the server or on the client. Generally, XSS on the server is easy to test using e.g. code analysis while testing for XSS on the client is hard.
- **Insecure Direct Object References** - When exposing a direct object reference to the user, without any access control checks. This means that a user can modify the reference to access other, possibly unauthorized data. Some examples of this could be when a file, directory or user id is part of the URL. These flaws can easily be tested by manually changing parameter values, e.g. by changing `presentation.ppt` to `employees.xls` in `open.php?file=presentation.ppt`.
- **Security Misconfiguration** - Some software comes with default accounts and sample files, such as configuration files, applications or scripts. If these are left unattended they may provide means to an attacker to be able to bypass authentication or access sensitive information. Some systems also come with debug information turned on, which could be of great help for an attacker. These flaws can be tested efficiently by automated tools that can detect use of default accounts, unused services or vulnerable sample files.
- **Sensitive Data Exposure** - This happens when an application does not protect sensitive data sufficiently, such as passwords, session IDs, or credit card numbers. The most common such flaw is to not encrypt sensitive data both at rest and in transit. This vulnerability enables attackers to conduct crimes like identity theft or credit card fraud. It is easy to test whether sensitive data is encrypted or not, both in transit and at rest. It is harder to test if the encryption used is "strong enough" since strong is a relative measure in this case.
- **Missing Function Level Access Control** - When the functions of an application are not protected enough. This happens if an application fails to

2.4. WEB APPLICATION SECURITY

perform an authentication or authorization check for a specific function or e.g. if the application unintentionally exposes an internal function via a user controlled parameter. This is often the result of bad underlying application logic. As with insecure direct object references these flaws can easily be tested by manually changing parameter values, e.g. by changing 123 to 124 in `info.php?account=123`.

- **Cross-Site Request Forgery (CSRF)** - When an attacker forces a victims browser to perform a forged HTTP request. This request is controlled by the attacker and sent by the victim, containing any automatically included authentication information. A vulnerable application will see the forged request as a valid request from the user. This flaw is easily testable by using either a code analysis or an automated approach.
- **Using Components with Known Vulnerabilities** - A system or application usually uses multiple third party libraries, frameworks, open source software, or other components. Each of these components can potentially have a flaw that makes them vulnerable, which in turn can compromise anything that uses that component. Using components with known vulnerabilities in an application may therefore enable a large range of possible attacks. Generally it is easy to test this flaw by checking all components against a database that contains vulnerable component versions.
- **Unvalidated Redirects and Forwards** - When an attacker can redirect a user to an untrusted site using functionality at a trusted site. This happens when the application uses untrusted data to determine the destination page of a redirect. This vulnerability exploits the trust a user has in a site and enables attacks such as phishing. Unvalidated redirect flaws can be tested easily by identifying all URLs where it is possible to specify a full URL, while unvalidated forwards are harder to test since they target internal pages.

Relevance for Nordnet

As mentioned earlier, the overall security concern for Nordnet is to protect their customers, and their customers trust (see [section 2.1](#)). Therefore the following items are important to protect:

- Any applications that a customer can interact with.
- Stored and potentially sensitive customer data.
- Customer assets (money and holdings).

In terms of vulnerabilities, any vulnerability that has a direct impact on Nordnet can be seen as more important since it affects the trust of their customers directly. The vulnerabilities with direct impact according to Nordnet are: **Injection, broken**

authentication and session management and **sensitive data exposure**. However, flaws in any of the ten times described above can lead to severe consequences, and hence all of them are important to Nordnet. The security concerns of Nordnet can be described in terms of the vulnerabilities in the list above:

- Three of the vulnerabilities directly affects user trust: **Cross-site scripting (XSS)**, **cross-site request forgery (CSRF)** and **unvalidated redirects and forwards**.

XSS and *unvalidated redirects and forwards* exploit the trust that a user has in a website. In the case of XSS this leads to malicious code being executed on a trusted site and in the case of unvalidated redirects a trusted site is used to send a user from a trusted site to a malicious site. A CSRF vulnerability exploits the trust that a user has in its browser. However, if a trusted website is vulnerable to CSRF, the trust that the user has for that website is affected as well. All of these three vulnerabilities can use social engineering to succeed, often to trick the user to click a link crafted by an attacker.

- Five of the vulnerabilities specifically deals with getting access to different types of sensitive data: **Broken authentication and session management**, **insecure direct object references**, **security misconfiguration**, **sensitive data exposure** and **missing function level access control**.

If sensitive data is leaked it damages, or possibly breaks the user's trust, which means that these vulnerabilities also affects trust, but indirectly. Broken authentication and session management relies on finding sensitive data to get further into a system, insecure direct object references and missing function level access control can both be used to access sensitive data, sensitive data exposure is self-explanatory and security misconfiguration can cause one of the four other vulnerabilities. All of these five vulnerabilities can be seen as stepping stones that enable an attacker to get further into a system.

- The two remaining vulnerabilities are: **Injection** and **using components with known vulnerabilities**.

Using components with known vulnerabilities is never a good idea since this can cause other vulnerabilities, including the others in OWASP top 10. Using such components may also be seen as a sign of ignorance, which also affect trust. Injections are one of the worst kind of attacks against web applications since an attacker using injections may not only get access to sensitive data, but she may also corrupt or crash systems.

2.4.2 Security testing approaches

Tied to each category of vulnerabilities in the OWASP top 10 project is some additional information on how an attacker can exploit them, what kind of impacts that exploit can have on a technical and a business level, and most importantly,

2.5. AGILE SECURITY TESTING

how such an exploit can be prevented. Apart from these there are also some more general guidelines on how to achieve a secure SDLC with security testing [44].

There are a few different ways to approach security testing. The most common approaches to identify vulnerabilities in an application are either by looking at its source code, or by analyzing it while it runs. A third alternative is to combine the two approaches, and therefore the three biggest categories of Application Security Testing (AST) are [21]:

- **Static Application Security Testing (SAST)** - Security specific static testing. Tests the source code of an application by trying to identify bad code or bad coding practices. This is sometimes called source code analysis or review [44].
- **Dynamic Application Security Testing (DAST)** - Security specific dynamic testing. Tests the application as the customer or consumer sees it, by trying to breach or break it. This is sometimes called penetration testing since the goal is discover potential vulnerabilities by penetrating the defenses of the application [44].
- **Interactive Application Security Testing (IAST)** - A combination of SAST and DAST that can both look at the source code and verify any findings by performing an attack on the application directly [21].

2.5 Agile security testing

Due to the recent explosion in vulnerabilities shown in [Figure 2.6](#), security testing has become the most important type of testing. Organizations have started to regard security differently since they have realized that the money they could save by eliminating vulnerabilities outweighed the money that they could earn by developing a new feature [55], as suggested in [section 2.3](#). At the same time the web is growing, and with it the number of web applications made within an agile development process. This poses a conflict since security testing is often a time consuming and a quite tedious process that relies heavily on manual verification, and therefore it is not well suited for the Agile development process. As presented in [section 2.2](#), there are data suggesting that the Agile method is superior to the most well-used traditional software development method; the Waterfall method. Yet, many companies still hesitate to adopt the Agile method due to its lack of security; security is not even mentioned in the Agile Manifesto [7]. A big problem is that we are used to a security process adapted for software development processes based on the Waterfall method, which means that it is rigorous and slow. Such a process can not possibly be introduced in the Agile development process since it goes against the Agile Manifesto. What is more, Agile focuses on partial deliveries, and security traditionally needs to be tested on a finished product [55]. Therefore, security must be dealt with differently to fit in the Agile development process.

Many have tried to introduce security in the Agile development process, but so far there is not much that has caught on. There are studies suggesting that security can be included as early as in the requirements phase by adding what they call misuse stories [32] to the original user stories. These misuse stories are designed to capture malicious use of the application and hence they can catch potential vulnerabilities already in the requirements phase. This suggestion looks promising, especially considering how much money there is to be saved according to Figure 2.5. However, authors of misuse stories and developers that need to implement them need to have a great deal of security knowledge for this to work out, which is not likely the case. Other studies reach conclusions that are highly focused on their particular conditions [59], which sadly helps very little in other settings.

Research about a secure agile development process is just getting started with the *First International Workshop on Agile Secure Software Development* being held later this year (Aug 2015) [3]. A recent presentation from a member of OWASP [46], suggests some novel ideas on how to achieve security in the Agile development process [11]. It is suggested that security can be integrated in the automatic build process by performing either *source code analysis* or *automatic security testing* using a vulnerability scanner, which can be seen in Figure 2.7. Other sources also suggest using tools (source code analysis tools or vulnerability scanners) to get security into the development process [36].

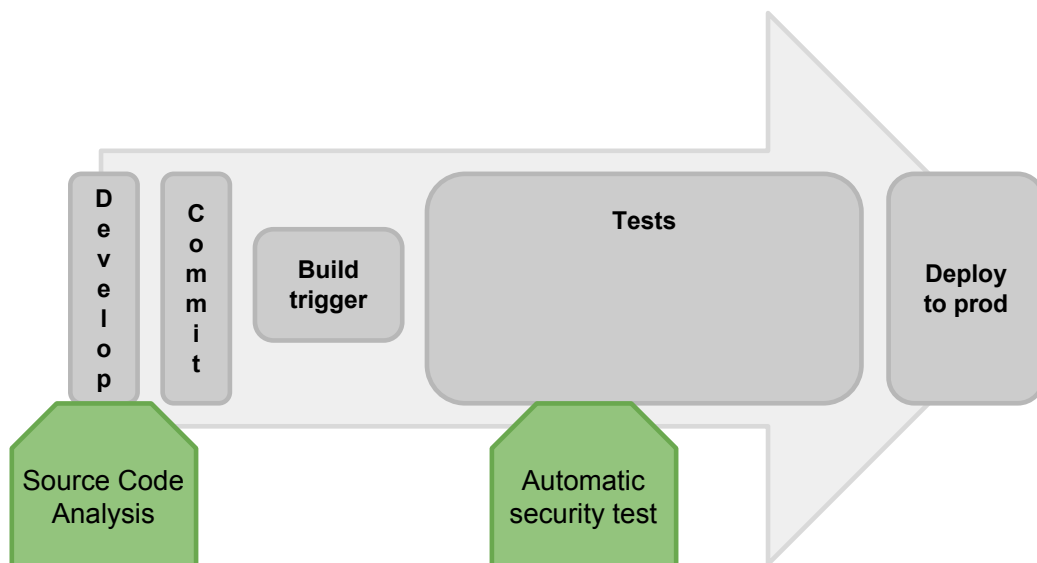


Figure 2.7. Suggestions on where and how to integrate security in an automatic build process [11]

These suggestions have been made possible since security testing is catching on to the automation trend of software testing (see subsection 2.3.2). Some of the most popular manual scanners used for penetration testing have recently been upgraded

2.5. AGILE SECURITY TESTING

to support automated scans [51, 8] and new fully automatic scanners have begun popping up in the last few years [1, 33]. Both suggestions in [Figure 2.7](#) are theoretical but a company called Continuum Security [64] has come a long way while trying to develop and implement approaches based on these. However, the approaches presented by them so far require quite some work to use, and test cases are still required to be written, which means that a great deal of security knowledge is still required.

The main components of the two suggestions mentioned in [Figure 2.7](#) are presented and discussed in greater detail below. These two are: source code analysis tools and vulnerability scanners. Their advantages and disadvantages will also be addressed with a focus on web applications, since that is the main focus of Nordnet.

2.5.1 Source code analysis tools

Source code analysis tools are tools used to perform SAST, i.e. analysing software source, or sometimes binary code for known vulnerabilities [36, Chapter 4]. The tools considered for this thesis are automated, to fit the Agile method, and have support for web application languages, since that is the main focus of Nordnet. Source code analysis tools typically consist of multiple components: one component that looks for common vulnerabilities by using methods such as taint analysis and data flow analysis [28], another component that checks the code style, e.g. PMD [50], and a third component that will try find common bugs, such as FindBugs [61]. The methods used by a source code analysis tool are usually derived from compilers and hence can be seen as an additional compiler in the development process since it will also parse and analyze the source code, even though it does not produce any binaries. These tools analyze the code by trying to match it logically or lexically against specific patterns and rules from a predefined list. This makes it possible to adapt source code analysis tools to a specific system and it often means that the analysis is fast, although this also depends on the number of checks to be performed and what type of methods are employed. A source code analysis tool should be executed regularly [36, Chapter 4] in the development stages of the process, either directly on the developers machine or in the build process.

The main advantages of using source code analysis tools are that they scale well, they give very fast feedback [36, Chapter 4] and they can generally handle many problems as discussed above. Since most errors are independent of the kind of software that is developed, and more dependent on the code these tools scale well and can be run on a lot of different software. Since they can be run frequently and repeatedly once set up, and since they can run on an unfinished application they can give very fast feedback to developers, which can mean saving a great deal of money as shown in [section 2.3](#). As discussed above a source code analysis tool can deal with many different types of problems, such as finding bugs, checking code style and detecting common vulnerabilities. This can be seen as a disadvantage if the tool becomes too unfocused. Tools that are able to deal with many types of problems often look very tempting to use but they are often less effective than specialized

tools solving the same problem, so this could be a potential disadvantage for the source analysis tools.

The main disadvantages of using source code analysis tools are that they are highly dependent on what programming language is used and what code style a certain programmer has, and that they have a hard time proving if an identified issue is an actual vulnerability. All these disadvantages manifest themselves through the false positive rate of the source code analysis tool, which is usually very high [28]. Creating a set of rules that all developers can agree on and follow is hard since code style is typically heavily opinionated, and such an endeavour may take a long time since there are a lot of settings to tune.

2.5.2 Vulnerability scanners

Vulnerability scanners are tools used to perform DAST, i.e. scanning software for known vulnerabilities [36, Chapter 6]. The scanners considered for this thesis are automated, to fit the Agile method, and are used to scan web applications, since that is the main focus of Nordnet. These are called web application vulnerability scanners and they generally consist of two components: a *crawling* component and a *scanning* component. The crawler is executed first and its goal is to map the web application. It is initialized with some initial URLs (endpoints) for a web application, which it will visit while recording new URLs and potential points where data can be injected, such as GET parameters or HTML forms. The scanning component is further divided in to two modes; *active* and *passive*. Active scanning is sometimes also called penetration testing and its purpose is to actively penetrate a web application while analyzing its response. It will typically be executed after the crawling is done since it uses the information found in the crawl, but it can also run alongside the crawl, attacking the web application at the same rate it is discovered. Passive scanning on the other hand is when the scanner observes use of an application and tries to detect unsafe practices, such as trying to list directories and identifying any unsafe technologies used. A passive scan will typically run during the crawl by investigating the requests.

The main advantages of using a vulnerability scanner are that they are easy to use, they have a high level of automation and they are able to scan a web application independently of how it is implemented. A scanner can scan multiple web applications for known vulnerabilities with the click of a button, and it is typically much faster than a human trying to find vulnerabilities manually. However, the biggest limitation of vulnerability scanners are that their false negative and false positive rates are high compared to manual testing performed by a human. These high rates have been observed in various previous benchmarks [13, 14] and it seems like they are highly coupled with the scanners ability to crawl [17].

Vulnerability scanners target known vulnerabilities, just like antivirus software target known viruses. This is an advantage when it comes to regression testing but it also means that they are unable to find *new* vulnerabilities, and that is one of the reasons for their relatively high false negative rates. Scanners rely on their

2.5. AGILE SECURITY TESTING

creators knowledge and skill, and on a steady stream of updates to their detection abilities to be able to keep the false negative rates at bay while being able to detect the latest vulnerabilities. For scanners, certain types of vulnerabilities are harder to detect than others since a scanner cannot foresee every possible variation of a vulnerability. Many scanners will therefore mark cases of which it is unsure as potential vulnerabilities that require further verification by a human. Historically, one major limitation of vulnerability scanning is that it usually represents a too-little-too-late attempt at security since it is performed at the end of the SDLC [36, Chapter 6].

One thing to watch out for when testing with web application vulnerability scanners is that since they test over the web they are limited by the network between the scanner and the web application, and any other connections that the web application might have. The network can inflict random errors during testing that can sometimes give misleading results, and although many popular scanners have countermeasures for such potential issues it is something to watch out for when testing.

Chapter 3

Approach

This chapter presents the methods used to investigate the problem posed in [section 1.2](#), and the reasoning behind why these methods are chosen. This includes a pre study containing an evaluation of the background and an interview study, a section on evaluating vulnerability scanners, a section on testing vulnerability scanners, a section on integrating and implementing one of the scanners in the delivery pipeline at Nordnet (see [Figure 2.4](#)), and finally a section about producing a guide for Nordnet.

3.1 Pre study

To investigate the problem posed in [section 1.2](#) an evaluation of the material in the background, targeting agile security testing has been performed. After this an interview study has been planned and conducted to find out the requirements of Nordnet.

3.1.1 Evaluating approaches for agile security testing

Two proposals on how to integrate security in the Agile development process are presented in [section 2.5](#); using a source code analysis tool or using a vulnerability scanner. Neither of these two proposals are sound, which means that although both of them are designed to find vulnerabilities there is no guarantee that the analysis tools or scanners can find all vulnerabilities. For each of the proposals there is a section explaining them in greater detail ([subsection 2.5.1](#) and [subsection 2.5.2](#)), which contains some general information about how they work and what their strengths and weaknesses are.

Comparing the two approaches, source code analysis yields a lower false negative rate overall, which is desirable, but it usually also yields a higher false positive rate. This means that source code analysis requires more manual verification than automatic security testing with a scanner. To integrate security in the Agile development process, security must itself become agile as addressed in [section 2.5](#), and

since manual verification hinders automation the use of a scanner is a more suitable approach. Web applications can be built using different styles, frameworks and programming languages. Automatic security testing with a scanner can test all types of web applications, since a scanner is independent of what programming language is used whereas source code analysis is not. Each source code analysis tool supports a handful of programming languages at most, which can make it impossible to find a tool that can test all web applications at Nordnet. This means that employing automatic security tests by using a scanner is the best approach in this situation as well. Overall, this comparison implies that a web application vulnerability scanner is better suited for the context of developing secure web applications using the Agile model at Nordnet.

In [Figure 2.7](#), it is proposed that automatic security testing with a vulnerability scanner can be integrated in the testing phase in an automatic build process. The process is different on Nordnet (see [Figure 2.4](#)), and to find a good place to integrate a scanner in their delivery pipeline more information is needed. Additional information about what vulnerabilities are important to Nordnet is also needed since scanners come in many different flavours that target different vulnerabilities.

3.1.2 Interview study

An interview study has been conducted in order to obtain information about the security requirements specific for Nordnet, and to determine the general security knowledge at Nordnet. Additionally, the interviews are used to find out what the requirements are for a method to fit in the Agile development process at Nordnet, i.e. any information needed to choose a scanner and decide how and where to integrate it in the delivery pipeline at Nordnet.

The interviews has been performed according to the semi-structured method. According to this method a set of themes or a template of open ended questions are first created and later adapted to each interviewee during the interview [5, 27]. This allows for planning and conducting interviews with people of different backgrounds and experience. The semi-structured interview is very much controlled by the experience of the interviewee, and this means that the *interviewer effects* can be reduced [16].

Since security affects everyone in the development process, the target group for the interview consists of developers (main target group), quality assurance (QA) engineers and other people involved in the delivery pipeline.

Developers are in the target group since they are the main target for this thesis. Ultimately it is they who will have to modify their projects and set everything up so that the automated scanning can run. It is also they who will see the results of a scan first, and they who will have to verify and fix most of the issues that are found while scanning. There is not enough time to interview all developers, and therefore interviews has also been conducted with some people with the scrum master role. Since they lead teams of developers they can hopefully help with estimating the security knowledge of their teams.

3.2. EVALUATING SCANNERS

QA engineers are in the target group because it is their responsibility to ensure quality of software, which also involves security. This means that they have the final say if an issue must be fixed or if it can be ignored (e.g. a false positive [48]). They are also highly involved in the overall delivery pipeline and as such they are possible users of the scanner, its outcome and/or the guide.

The rest of the people in the target group are responsible for the delivery pipeline. Since the scanner will probably be integrated somewhere in that delivery pipeline, and since they decide what goes or not, their suggestions and insights might be valuable.

The interview study was used in order to find out:

- **The level of security knowledge of the target group** - To know what *level* the security information should have, which will be included in the guide.
- **What security requirements and concerns that Nordnet have** - To pinpoint what type of vulnerabilities and security risks that are important to Nordnet and how they rank these.
- **How Nordnet works with security today** - To find out positive and negative aspects in their current way of working with security.
- **What systems / frameworks / programming languages are used at Nordnet** - To get requirements on what types of vulnerabilities that a scanner should be able to test for.
- **How security can be integrated in the Agile development process at Nordnet** - To find constraints and possible suggestions for this work for the specific implementation of the Agile development process at Nordnet. Since they are the experts on their process they will have valuable insights to share.
- **What expectations the interviewees have on a possible future solution** - To get additional information on where in the process a security solution could be integrated, and how it could be implemented.

3.2 Evaluating scanners

There are many web vulnerability scanners available on the market [14], and the goal of this section is to reduce that amount to something manageable. To do this, requirements for a scanner specific to Nordnet has been produced. These requirements, and the requirements obtained from the interview study in [subsection 3.1.2](#), were then used to compare scanners.

Evaluations and benchmarks of scanners often express the capabilities of vulnerability scanners by what attack vectors they support [13, 14]. Comparing scanners based on which attack vectors they support is a good way to quickly sift through a big number of scanners. However, a weakness with this type of comparison is that attack vectors can be supported to different degrees. If a scanner supports an attack

vector, it simply means that the scanner might be able to find some vulnerabilities connected to that vector, but maybe not all. Since evaluating scanners is not the main contribution, or main goal of this work, that weakness has been disregarded in this initial sifting of scanners, and handled in the next section where scanners are tested.

3.2.1 Gathering requirements

In order to find what requirements are important for a scanner, two lists of important attack vectors has been compared to the security concerns of Nordnet (gathered from [subsection 2.4.1](#) and the results of the interview study in [subsection 3.1.2](#)). As mentioned in the background chapter, a vulnerability scanner can not test all vulnerabilities described in OWASP top 10 (see [subsection 2.4.1](#)), e.g. a web application vulnerability scanner can never determine if user credentials are stored securely in a database. Therefore, the requirements has been based on one list that contains *“a base set of vulnerabilities that a web application security scanner must be able to identify if it supports the technology and languages in which the vulnerability exists”* [9]. However, since that list of attack vectors is quite old (published in 2009), another list of important attack vectors from a recent benchmark evaluating 63 scanners was also used [14]. Using both of these ensures that the requirements consist of both basic and recent attack vectors. These attack vectors were then checked against the security requirements of Nordnet to create a list of requirements for a scanner adapted to Nordnet.

3.2.2 Comparing scanners

As mentioned above, there are many web vulnerability scanners available on the market [14]. 20 scanners has been picked out from a recent benchmark [14] to obtain a starting point for further comparison. These scanners are chosen according to the number of attack vectors they support in that benchmark, since a scanner that supports many attack vectors has a higher chance to fit in this context. After that, the list of scanners is reduced based on the requirements from the previous subsection, and on the following:

- **That the scanner has support for the technology at Nordnet** - This is important since the scanner will ultimately be used at Nordnet. The scanner should support their input methods (such as HTTP GET, XML, etc.) in order to communicate with their web applications. It should also support their authentication method(s) so that web applications that require authentication can be scanned.
- **How the scanner handles the general problems with vulnerability scanners mentioned in [subsection 2.5.2](#)** - Handling these problems could mean: having a lower false positive rate, having a lower false negative

3.3. TESTING

rate, being able to detect new vulnerabilities or being better adapted to new technology.

- **How much the scanner costs** - This is important since a scanner can cost a great deal, both to buy and to use. The scanner may come with an expensive payment model, it may be unable to find vulnerabilities that can be exploited by attackers and therefore incur costs or it can be poorly adapted and therefore have a high maintenance cost.

3.3 Testing

Since the plan is to integrate and implement *only* one scanner, the reduced list of scanners from [subsection 3.2.2](#) has been tested and compared more thoroughly. Both their crawling and detection abilities has been tested, and the latest version of each scanner was used when testing.

3.3.1 Environment

To test the scanners in a reliable way, they have been compared on deliberately vulnerable applications [15, 62, 35]. Such applications are designed to be vulnerable and they have a fixed set of vulnerabilities, which is helpful since it allows easy tracking of false negatives. Some of these applications also provide false positive test cases that can be used to test the accuracy of scanners. A general drawback of using deliberately vulnerable applications is that they are publicly available, which means that vendors of vulnerability scanners can *cheat*, and adapt their scanners to perform well on those applications [17]. However, this form of cheating can be avoided when testing, by carefully picking the applications so that they support a wide range of vulnerabilities. Because, if they support a wide range of vulnerabilities they can be seen as general enough to represent *most* applications. If a scanner performs well when scanning such an application, it will probably also perform well on other applications, even if it was adapted to perform well on that specific application.

Another drawback when using deliberately vulnerable applications when testing scanners, is that they are designed as training grounds for humans (e.g. WebGoat [35]). Such applications typically contain hints and systems to keep track training progress, which makes them less than ideal when testing automated vulnerability scanners. Luckily, there are also deliberately vulnerable applications specifically designed to evaluate scanners on: the Web Input Extractor Teaser (WIVET) [62] and the Web Application Vulnerability Scanner Evaluation Project (WAVSEP) [15] are two examples. Since a vulnerable application contain a fixed set of vulnerabilities, as mentioned above, they will not necessarily fit the all the security concerns of Nordnet, and therefore the scanners have also been tested on applications provided by Nordnet. This will not only make sure that the scanners work on Nordnet, but it also compensates for the potential cheating mentioned above.

WAVSEP and WIVET are both used in a previous benchmark [14], but the results still needs to be verified since many of the scanners have been updated since. Rerunning the tests also gives greater insight into how different scanners work, and how they can fit in the development process, which is investigated in the next section. This thesis therefore contributes new data, and since testing were also performed on real applications at Nordnet, the testing results can also be used to estimate the accuracy and robustness of WAVSEP and WIVET.

WIVET

WIVET [62] is an application designed to test the crawling ability of a scanner (see [subsection 2.5.2](#)), i.e. how well a scanner performs when trying to discover its target, which it does by finding input vectors (e.g. URLs). This is important to measure since finding more input vectors increases the attack surface of the scanner. WIVET v4 has been used for testing since it is the latest version available when this is written.

WIVET includes a broad array of crawling tests (56 different) and it has logic for keeping track of the statistics for each scan. Since WIVET uses a session cookie to keep track of ongoing scans [62], the scanner must have some kind of session handling to run WIVET successfully.

WIVET is included in this test because crawling is a critical ability for any scanner, and it has a big impact on overall performance [17]. A scanner without any crawling abilities would have to be provided with all valid URLs within the application being scanned, and that is not feasible since a list of all valid URLs must be regenerated / modified each time the application changes.

WAVSEP

WAVSEP [15] is an application designed to test the detection abilities of a scanner, i.e. how many of the known vulnerabilities it can find, how many it misses (false negatives) and how many of the false positive test cases it reports as vulnerable. Testing the detection abilities of a scanner will inevitably also test its crawling abilities (see [subsection 2.5.2](#)), which means that testing on WAVSEP is also a good complement to testing on WIVET. WAVSEP contains a large collection of test cases for different categories of vulnerabilities (see below and [subsection 2.4.1](#)). WAVSEP was created by Shay Chen [60] for his evaluation of web application scanners. WAVSEP v1.5 has been used for testing since it is the latest version available when this is written.

WAVSEP includes the following test cases (contains cases that use both HTTP GET and HTTP POST) [15]:

- **Local File Inclusion/Directory Traversal/Path Traversal** - 816 test cases and eight categories of false positives.
- **Remote File Inclusion** - 108 test cases and 6 categories of false positives.

3.3. TESTING

- **Reflected Cross Site Scripting (RXSS)** - 64 reflected test cases, 4 DOM based cases and 7 categories of false positives.
- **SQL Injection** - 80 error-based test cases, 46 blind test cases, 10 time based test cases and 10 categories of false positives.
- **Unvalidated Redirect** - 60 test cases and 9 categories of false positives.
- **Old, Backup and Unreferenced Files** - 184 test cases and 3 categories of false positives.

WAVSEP is included because it has a large number of test cases in different vulnerability categories, and it is designed for the specific purpose of evaluating vulnerability scanners. All the vulnerabilities supported by WAVSEP, except for the two relating to *file inclusion*, are featured in the OWASP top 10 list, and therefore important to Nordnet (as discussed in [subsection 2.4.1](#)). However, the remaining two vulnerabilities can be seen as members in the categories *security misconfiguration* and *sensitive data exposure*, and as such they are also important to Nordnet.

Since there is data available from old tests [14], they can be used to verify the testing results. Access to old test data is important in this case because it can help reduce potential cheating of vendors mentioned above. Since WAVSEP v1.5 was not publicly available before the previous benchmark it *probably* also means that the results from it are unbiased.

Nordnet

To make sure that the results from WIVAT and WAVSEP are reliable, and applicable to Nordnet, scanners have also been tested on applications provided by Nordnet. A great deal of the results in this thesis are ultimately going to be used at Nordnet and therefore it makes sense scanning there as a final test. As mentioned in the background (see [section 2.1](#)), Nordnet already scans their applications in test and production environments, using both automated and manual scans. These scans will find vulnerabilities that can be used to test the remaining scanners on. Testing on known vulnerabilities helps with catching potential false negatives reliably.

3.3.2 Execution

The testing has been performed in a virtual machine environment. Two virtual machines has been set up: one that houses the scanner (Host S) and one that house the applications used to test the scanners on (Host T). The specifications of those machines are:

- **Host S**
 - **OS** - Ubuntu 14.04.2 LTS
 - **Architecture** - i686

- **Memory** - 3 GB
- **Processor** - Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz
- **Host T**
 - **OS** - Red Hat Enterprise Linux Server release 6.6 (Santiago)
 - **Architecture** - x86_64
 - **Memory** - 1 GB
 - **Processor** - Intel(R) Xeon(R) CPU X7460 @ 2.66GHz

These specifications are not important for the testing since raw performance is not measured, and therefore e.g. time is not a critical aspect. They are supplied for reproducibility.

The scanners have first been tested on WIVET, then on WAVSEP, and lastly on applications provided by Nordnet. How each of these tests has been carried out is presented below:

WIVET

To ensure correct measurements on WIVET, each scanner has been experimented with, and any features that can be used to increase their score has been used. This is done to achieve the highest possible score, and to make sure that the comparison between the scanners is fair. When a good configuration has been found, WIVET has been scanned three times by each scanner to help catch potential inconsistencies. Inconsistencies can be caused by e.g. network troubles as mentioned in [subsection 2.5.2](#). Since WIVET uses session cookies to track and rank each scan, each scanner also needs a way to set a session cookie. To simplify the configuration of each scanner, the source code of WIVET has been modified slightly by removing some of the logout links (100.php). This is done so that a scanner will not *click* such a link by mistake, since that will cause a session reset, and WIVET will no longer be able to track that scan.

WAVSEP

A similar approach to the one described above has been used when scanning WAVSEP. The following has been done to ensure that each scanner does as well as possible:

- If a scan does not identify all vulnerabilities in a category on the first try, the scan was restarted. This was done to help catch potential inconsistencies, as mentioned in [subsection 2.5.2](#) and in the WIVET section above.
- Each subcategory was scanned one at the time to avoid overloading the server, and to make the scans faster and easier to reproduce.
- Each scan was configured to only have the applicable checks for the specific test case active, to keep the scans fast and to avoid any irrelevant findings.

3.4. INTEGRATION AND IMPLEMENTATION

Some test cases has been removed from the scan since they are not supported by the host, or not fully working:

- Path Traversal/LFI test cases 10, 12, 14, 16, 18, 20, 24, 37 and 54 were removed since they all require WAVSEP to be hosted on a Windows machine (they exploit the way Windows builds paths), but it has been hosted on *Host T*, which runs Linux.
- Experimental test cases in all categories, except the DOM based XSS cases, were removed since they are unstable/not fully working.

Nordnet

When testing on applications provided by Nordnet a similar approach to the one described above (for WAVSEP) has been used. Each scanner were run against each vulnerability found on Nordnet, while only enabling the applicable checks or options. Due to potentially sensitive data in this test, exact details of the vulnerabilities is not given out, but rather some general indicators, such as the category of the vulnerability.

3.4 Integration and Implementation

When a scanner has been found, it must be integrated in the delivery pipeline. To be able to integrate and implement a scanner in the current development process at Nordnet (described in [Figure 2.4](#)), three places in the delivery pipeline has been identified, and then compared with the help of the requirements and wishes that were obtained from the interview study (see [subsection 3.1.2](#)). Once a place is chosen, a scanner has been implemented to seamlessly fit in the development process.

3.4.1 Integration

“As a measurement tool, penetration testing is most powerful when fully integrated into the development process in such a way that early-lifecycle findings are used to inform testing and that results find their way back into development and deployment practices.” [[36](#), Chapter 6]

Three places where a scanner can fit in the delivery pipeline has been investigated and compared (see [Figure 3.1](#)). A presentation of these three can be found below, containing a description of how security testing can fit there:

- **Before committing** - The security testing is performed just before the delivery pipeline, in [Figure 3.1](#), takes place. This means that each developer will have full responsibility for testing the code before committing it.
- **In the CI: Automatic tests step** - The security testing is performed in the automatic test step of [Figure 3.1](#).

- **In the QA: Tests step** - The security testing is performed during the QA testing step in [Figure 3.1](#). Some scanners have the ability to act as a scanning proxy [17], which can be used for automated QA testing with e.g. Selenium [29] or manual QA testing with a browser by directing traffic through the scanner.

These three places has been evaluated using the requirements that were obtained from the interview study in [subsection 3.1.2](#). Each of the places has been analyzed by looking at pros and cons before deciding where to implement a scanner. After deciding where in the development process a scanner will be integrated, any cons with that suggestion were further investigated before moving on with the implementation.

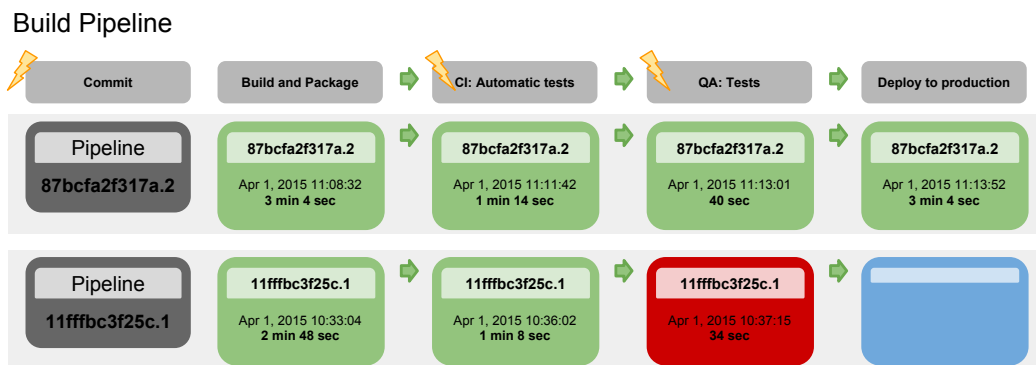


Figure 3.1. A version of [Figure 2.4](#), where the delivery pipeline at Nordnet is extended with markings (in the shape of flashes) showing the three places where security testing is possible

3.4.2 Implementation

To implement security testing in the Agile development process it must be automated, and therefore a scanner has to be seamlessly integrated. This probably means that some kind of wrapper around the scanner is needed to set up the target(s) and to specify settings for the scan. Since it is possible to choose what settings and attacks a scanners should use, a default configuration has been produced, tailored to the needs of Nordnet. The wrapper also needs to make sure that some kind of scan report is created so that the developers and QA engineers can see how the scan went, and react accordingly. According to the web application security scanner specification [9], a scan report should contain the following information:

- **Mandatory items**
 - A description of an attack that demonstrates the issue.
 - Further specifying that attack by providing script location, inputs and context.

3.5. GUIDE

- A name identifying the vulnerability that semantically equivalent to the definitions in the specification [9].

- **Optional items**

- The severity of the issue.
- Remediation advice for the issue.

3.5 Guide

When this section is reached, a scanner has been chosen and integrated in the Agile development process at Nordnet. One of the goals for this thesis is to create a guide for developers, containing information about how they can use the implementation, and how using it will help them. The following topics has been addressed in the guide:

- How a developer can set up security testing on a project.
- How they should interpret any output from the security testing.
- How they can fix any vulnerabilities found during testing.

The guide has been adapted to Nordnet in both content and layout, to make it more familiar to the target group (the same target group as in the interview study). This was done by using document templates and other internal guides found on Nordnet. For inspiration about writing technical security related guides for the target group, two OWASP guides were used:

- The OWASP Developer Guide 2014 [43].
- The OWASP Testing Guide v4 [44].

Chapter 4

Results

This chapter contains all the results of this thesis. These are presented in different sections based on the sections in [chapter 3](#). The results are presented in summarized form, and in some cases (where noted) the full results can be found in one of the appendices.

4.1 Pre study

The interviewees ended up containing four developers, one scrum master, two QA engineers and two people responsible for the delivery pipeline. During the interview study it became clear that QA engineers and scrum masters had a broader view of security than developers, while developers generally had more information about detailed threats, like XSS and SQL injection (see [subsection 2.4.1](#)). This is not surprising since a developer's work includes fixing bugs, which includes security related bugs that require detailed knowledge. A QA engineer works with finding and verifying issues in software, which includes security related issues that require both general security knowledge, and some specialized knowledge. The level of security knowledge in the target group depends on the interaction between security testing and the development process. This is a common problem when security testing meets agile development as described in [section 2.5](#). As presented in the background (see [section 2.1](#)), security testing is performed both internally at Nordnet, and externally. None of these groups contain developers or QA engineers from Nordnet, and therefore security does not influence the development process much.

Even though there were no security experts in the target group, the interviewees still provided good insights regarding the security requirements at Nordnet. Many of the developers mentioned that they were familiar with the vulnerabilities XSS and SQL injection. This could mean that these vulnerabilities are common on Nordnet, or that the vulnerabilities have been around for such a long time that everyone knows about them. Either way, both of these scenarios show that such issues are important, but that does not necessarily mean that they are the *most* important vulnerabilities to Nordnet. Most of the security related concerns that came up during the interviews

were in alignment with the discussion in [subsection 2.4.1](#), and the overall impression was that all vulnerabilities in that list are equally important to Nordnet.

These are the requirements and wishes of the target group on how and where security testing can be performed in the Agile development process at Nordnet using a scanner:

- **General:**
 - The scanner can be interacted with from the command line.
 - The scanner should be *actively* maintained.
 - The false positive ratio should be *as low as possible*.
 - The false negative ratio should be *as low as possible*.
 - The scanner can handle basic authentication (can scan as a logged in user).
 - Reports should gather similar issues to avoid ambiguities.
- **From developers:**
 - Any reports should be easily understood for developers even if they are security novices.
 - Developers should not have to write any extensive tests (scanning should be as automated as possible).
- **From people responsible for the CI environment / Jenkins:**
 - The scan needs to be fast, maximum five minutes.
 - The scanner should preferably be able to run on Linux.
 - The scanner should indicate how the scan went by exiting with certain codes or by providing a scan report in JUnit [30] format.
- **From QA:**
 - The ability to somehow mark false positives and hide those in future scans.
 - The automated scan should not be able to fail the build on its own.
 - The ability to manually mark the build as *pass* or *fail* based on the scan report.

4.2 Evaluating scanners

In this section a list of attack vectors applicable to Nordnet is compiled, then 20 scanners are compared with the help of these requirements.

4.2. EVALUATING SCANNERS

4.2.1 Gathering requirements

As a foundation for the scanner requirements a list of attack vectors from a recent benchmark [14] are used. This list of attack vectors is then reduced to only contain important attack vectors that are applicable to Nordnet. To reduce the list, a web application security scanner specification from NIST [9] is used together with the requirements from the pre study (see [section 4.1](#)). The following list motivates why a particular attack vector is eliminated:

- These attack vectors rely on one or more technologies not in use by Nordnet:
 - Server-Side JavaScript (SSJS/NoSQL) Injection is eliminated since Nordnet does not use NoSQL databases.
 - XML Injection is eliminated since web applications at Nordnet do not use XML.
 - XPath/XQuery Injection is eliminated since Nordnet does not use XML, which XPath/XQuery exploit.
 - Xml External Entity is eliminated since Nordnet does not use XML.
 - LDAP Injection is eliminated since Nordnet does not have any web applications building LDAP queries.
 - SMTP/IMAP/Email Injection is eliminated since Nordnet does not have any web applications handling email.
- These attack vectors are not recommended by NIST for a security scanner [9], and they rely on items not available on Nordnet:
 - Buffer Overflow is eliminated since it typically targets the technology in browsers and web servers rather than web applications.
 - Integer Overflow is eliminated since it typically targets the technology in browsers and web servers rather than web applications.
 - Format String Attack is eliminated since it typically targets the technology in browsers and web servers rather than web applications.
 - Padding Oracle is eliminated since it is not controlled by web applications, but rather by the server and the browser.
- These attack vectors are eliminated based on other reasons:
 - JSON Hijacking is eliminated since a CSRF attack vector is included, which renders JSON Hijacking useless.
 - Application Denial of Service is eliminated since it is already tested in another, automated way at Nordnet.

The remaining attack vectors are applicable to Nordnet and they are presented in [Table 4.1](#). If it is unknown whether an attack vector is applicable at Nordnet, but the attack vector is important according to NIST [9], the middle column is marked with a question mark. A symbol is mapped to each attack vector for later reference: alphabetical symbols are used for applicable attack vectors and numeric symbols are used for the others.

Table 4.1. Important attack vectors for vulnerability scanners that are applicable to Nordnet (and some that probably are applicable). Each attack vector is also mapped to a symbol, which is used for reference in [Table 4.2](#) and [Table 4.3](#)

Attack vector	<i>Applicable at Nordnet?</i>	Symbol
Error Based SQL Injection	YES	A
Blind/Time-Based SQL Injection	YES	B
Reflected Cross Site Scripting	YES	C
Persistent Cross Site Scripting	YES	D
DOM Cross Site Scripting	YES	E
Path Traversal & Local File Inclusion	YES	F
Remote File Inclusion	YES	G
Unrestricted File Upload	YES	H
Open Redirect	YES	I
HTTP Header Injection & HTTP Response Splitting	YES	J
Code Injection	YES	K
Expression Language Injection	YES	L
Source Code Disclosure	YES	M
Cross Site Request Forgery	YES	N
Privilege Escalation	YES	O
Command Injection	?	1
Server-Side Includes Injection	?	2
Old, Backup and Unreferenced Files	?	3
Forceful Browsing / Authentication Bypass	?	4
Weak Session Identifier	?	5
Session Fixation	?	6

4.2.2 Comparing scanners

20 scanners from a recent benchmark [14] are mapped to the attack vectors defined in [Table 4.1](#). These can be seen in [Table 4.2](#). For each scanner, it is marked if the attack vectors are supported or not. The column titled # denotes how many of the preferred attack vectors the scanner has support for (there are a total of 21 vectors).

4.2. EVALUATING SCANNERS

Table 4.2. Scanners mapped to attack vectors applicable at Nordnet. This is a modified version of a table found in a recent benchmark of scanners [14], with attack vectors irrelevant to Nordnet are removed

Scanner	#	Supported attack vectors														1	2	3	4	5	6	
		A	B	C	D	E	F	G	H	I	J	K	L	M	N							O
IBM AppScan	20	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
WebInspect	20	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Acunetix WVS	18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✗	✓	
W3AF	17	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗	
Tinfoil Security	16	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	✗	✗	
Burp Suite Professional	14	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗	✓	✗	✓	✗	
NTOSpider	18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	
arachni	15	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗	✓	
ZAP	15	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✗	✓	
IronWASP	15	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✓	✓	
Syhunt Mini	11	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	
Syhunt Dynamic	11	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	
QualysGuard WAS	15	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✗	✓	✓	
SkipFish	11	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	
Wapiti	11	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✓	✗	✗	✓	
Netsparker	14	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	
ScanToSecure	14	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✗	
Jsky (Commercial Edition)	12	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓	
Sandcat Free Edition	11	✓	✓	✓	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	✓	✗	✓	✗	
Vega	10	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	

To reduce the number of scanners, the approach described in [subsection 3.2.2](#) is used:

- **Are the important attack vectors from [Table 4.1](#) supported** - According to the interview study, XSS seems to be a common vulnerability at Nordnet (see [section 4.1](#)), and it is currently in the third place of OWASP top 10 (see [subsection 2.4.1](#)). Three of the attack vectors (C, D and E in [Table 4.2](#)) are related to XSS, and therefore a scanner should preferably support as many of them as possible. However, since supporting an attack vector does not necessarily mean that all instances of that attack type will be found, this requirement is insufficient to warrant the removal of any scanner from the comparison. For future reference, the following scanners does not support the majority of the attack vectors concerning XSS (note that the format is: *Scanner Name (number of supported XSS attack vectors)*): Syhunt Mini (1), Syhunt Dynamic (1), Jsky (Commercial Edition) (1), Sandcat Free Edition (1) and Vega (1).
- **Is the technology at Nordnet supported** - Most of this is already handled by the previous criteria, but according to the interview study the scanner should *preferably* be able to run on a Linux machine. This requirement is insufficient to warrant the removal of any scanner from the comparison. For future reference, the following scanners can not run on Linux: IBM AppScan (Windows), WebInspect (Windows), Acunetix WVS (Windows), NTOSpider (Windows) and Netsparker (Windows).
- **Are general problems with vulnerability scanners handled** - One of the general problems with vulnerability scanners, mentioned in [subsection 2.5.2](#), is that a scanner needed regular updates to be able to handle new vulnerabilities. Therefore, this can be measured in activity, such as time from last release (and commit in the case of open source scanners). If the latest stable release of a scanner is over two years ago, or the latest commit is older than three months, that scanner is regarded as inactive. Therefore the following scanners are removed from further comparison (this was measured in May 2015): SkipFish (latest release and commit in December 2012), Wapiti (latest release in October 2013 and latest commit in November 2014), Syhunt Mini (latest release March 2012), Jsky (Commercial Edition) (latest release in February 2011), Sandcat Free Edition (latest release in April 2010) and Vega (latest release in July 2011).
- **How much does the scanner cost** - It is hard to know how good a scanner is before testing it, and therefore this section only handles raw costs, such as licensing and maintenance costs. Just like the requirement on technology above, this requirement is insufficient to warrant the removal of scanners from the comparison. However, an expensive scanner that does not support everything might not be worth considering. For future reference the following scanners are

4.2. EVALUATING SCANNERS

commercial: IBM AppScan (starting at \$10,400 per user per year), WebInspect (around \$10,000 per user per year), Acunetix WVS (\$3,995, consultant ed. 1 year), Tinfoil Security (starting at \$59 per month), Burp Suite Professional (\$299 per user per year), NTOSpider (around \$10,000 per user per year), Syhunt Dynamic (\$4,000 per user per year), Qualysguard WAS (starting at \$1,995 per year), Netsparker (starting at \$1,950 for 1 year) and ScanToSecure (unknown amount per scan). Since IBM Appscan, WebInspect, Acunetix WVS, NTOSpider and Syhunt Dynamic are all expensive, and have already appeared in this list they are removed from further comparison.

- **Does the scanner fulfill the requirements from Nordnet**
 - Since the security testing needs to be performed in an internal environment, the scanner must be provided as an *on premise* version, and not as *software as a service (SaaS)*. Therefore the following scanners are removed from further comparison: Tinfoil Security (they do offer to run scans through a VPN, but that will pose additional security risks), Qualysguard WAS and ScanToSecure.
 - One of the requirements obtained in the interview study is that a scanner can be interacted with from the command line. This is also one of the main requirements for this work since it enables programmatic interaction with the scanner, which is vital for automation. However, a scanner does not have to provide a command line interface (CLI) if it provides something else that enable programmatic interaction, like an API, a web interface, or something similar. Some of the scanners only provide a GUI version, and therefore the following scanners are removed from further comparison: IronWASP (no CLI, web interface, API or similar) and Netsparker (it does offer to run a scan via command line with arguments, but only a limited number of features can be accessed in that way, plus it has appeared two times before in this list).

Table 4.3. A version of Table 4.2 where only four scanners remain

Scanner	#	Supported attack vectors															1	2	3	4	5	6
		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O						
W3AF	17	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	✗
Burp Suite Professional	14	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗	✗	✓	✓	✗
arachni	15	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓
ZAP	15	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓	✗	✓

The four scanners in [Table 4.3](#) remain after the removal process described above. All four are active, they are affordable, and they fulfill the requirements from Nordnet. However, they also lack some of the important applicable attack vectors, which can be seen in [Table 4.3](#) and [Table 4.1](#). Most notable is that none of the four scanners have support for *expression language injection* (ELI). Recent vulnerabilities concerning ELI target an issue that can cause the Spring MVC framework [56] to *double resolve* expression language (it is evaluated twice), but the framework has since been updated to mitigate the issue. How these *missing* attack vectors affect the scanners performance can be seen in the next section. A short presentation of each of the remaining scanners follows:

Arachni

Arachni [33] is an open source scanner written in Ruby that consists of a framework and a web UI. Arachni features a rich CLI that has support for fully customizable scans, external plugins, generating scan reports in multiple formats and creating custom authentication scripts. Arachni also has support for running on several hosts in grid or cluster to leverage the scanning, which is unique among these scanners.

Burp Suite

Burp Suite [51] is a well-known commercial scanner. Burp Suite started out as a manual tool used in penetration testing and has since been upgraded to support several automated tasks. Burp Suite can be seen as a set of tools that support several different testing activities, e.g. a proxy mode where requests and responses can be investigated and modified, a scanner for automated penetration testing and a spider for crawling. Burp Suite also contains a marketplace for extensions called the *BApp Store*, where more options and features can be downloaded and added.

OWASP ZAP

OWASP ZAP [8] is an open source scanner written in Java. OWASP ZAP aims to be a scanner that can be used by people of varying background in security, such as developers or testers, but also by more experienced penetration testers. Like Burp Suite, OWASP ZAP features a proxy, a spider, a scanner and a marketplace for extensions. OWASP ZAP provides a REST API, which makes programmatic interaction easy.

w3af

w3af [52] is an open source scanner written in Python that provides two user interfaces, a console UI and a graphical UI. w3af is built around plugins, where the main three sets of plugins are *crawl*, *audit* and *grep*. Each plugin is configurable and they usually target a very specific threat or vulnerability. A notable long term goal

4.3. TESTING

of w3af is that it wants to “combine static code analysis and black box testing into one framework” [52].

4.3 Testing

This section contains information on testing the scanners above on deliberately vulnerable applications (WIVET and WAVSEP) and applications provided by Nordnet. It is divided into subsections based on the application tested against, and each of these subsections contains:

- A description of *how* each scanner was tested: how it was set up and what options were used.
- A summary containing the results of all scanners and some general observations made during the test.

The latest released version of each scanner is used for testing, with an exception for Arachni, where both the latest released version and the latest development version are used, since the latter contains many features and bug fixes [33]. The version used for each scanner can be seen in [Table 4.4](#).

Table 4.4. The scanners that are tested, their version

Scanner	Version
Arachni	v1.0.6 and v2.0dev
Burp Suite	v1.6.12
OWASP ZAP	v2.3.1
w3af	v1.6.48

4.3.1 WIVET

This section presents how each of the scanners in [Table 4.4](#) were tested on WIVET. A summary of the results is provided at the end of the section.

Arachni

Both versions of Arachni were tested in the same way on WIVET. After WIVET only v2.0dev is used because of performance enhancements, which can save a lot of time when testing the remaining applications. Note that <http://target/> is the URL of Host T in [subsection 3.3.1](#). To scan WIVET, the following command was used:

```
./bin/arachni http://target/wivet/ --checks trainer --
audit-links --audit-forms --scope-exclude-pattern=
logout --http-cookie-string="PHPSESSID=
h4pksk4dte915acu8sa8rnds11"
```

- The *trainer* check makes it possible for Arachni to *learn* from an ongoing scan, and adapt it accordingly.
- The *audit-links* and *audit-forms* options restrict the scanner to only audit links and forms (and not cookies, etc.). This is used to limit the scan, and thereby make it faster.
- The *http-cookie-string* option sets a cookie (in this case *PHPSESSID*) that Arachni uses while crawling, which lets WIVET track progress correctly.

Burp Suite

To scan WIVET with Burp Suite a proxy was first created and then a browser, using that proxy, was navigated to <http://target/wivet>. This made WIVET appear in the *Site map* of Burp Suite, and also set a session cookie in the *Cookie Jar* (under *Options > Sessions*), which is used by the *Spider*. Then, the *Scope* was established like in [Figure 4.1](#).

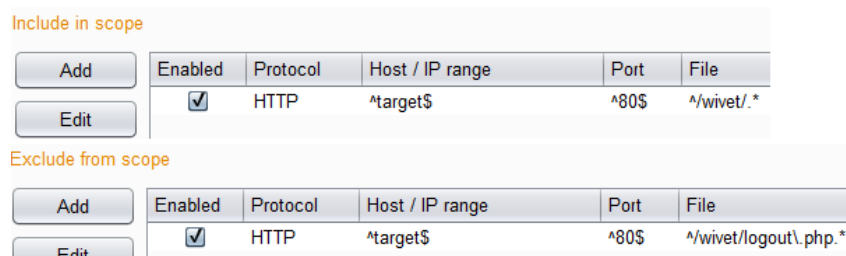


Figure 4.1. Scope settings of Burp Suite [51]

This means that Burp Suite follows all URLs that match <http://target/wivet/>*, except for the `logout.php` page. After setting up the scope, the WIVET folder (under <http://target/> in the *Site map*) was right clicked, and the option **Spider this branch** was chosen.

OWASP ZAP

To scan WIVET with OWASP ZAP a proxy was first created and then a browser, using that proxy, was navigated to <http://target/wivet>. This made WIVET appear in the *Sites* list of OWASP ZAP. Then, a *context* for the scan was set up like this:

4.3. TESTING

- Include in context: `\Qhttp://target/wivet\E.*`
- Exclude from context: `.*logout.*`

This means that OWASP ZAP follows all URLs that match `http://target/wivet/*`, except if they contain the word `logout`.

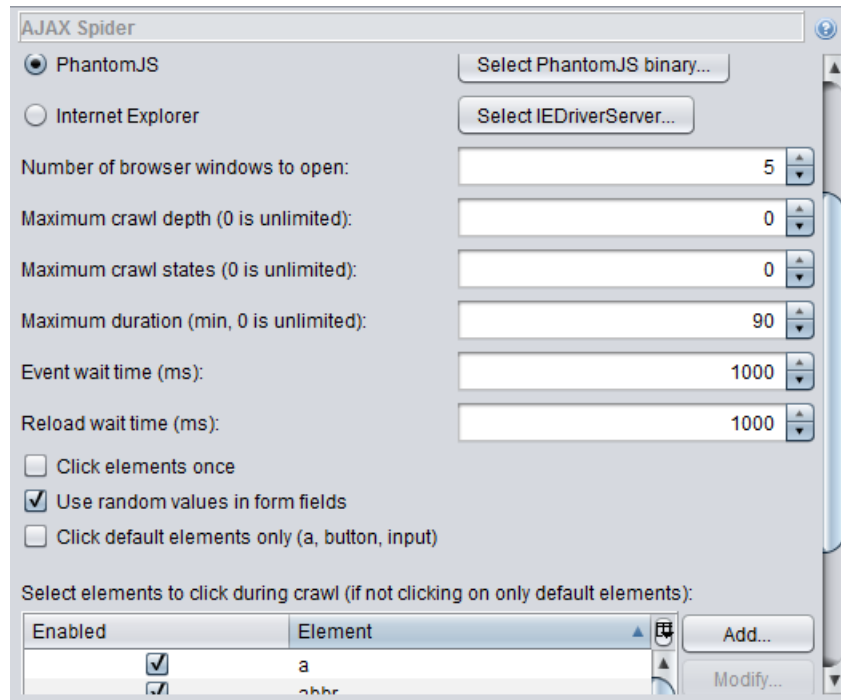


Figure 4.2. Settings window of the AJAX Spider [8]

After setting up a context, the AJAX Spider settings were edited as in Figure 4.2. Additionally, everything in the elements table (at the bottom of Figure 4.2) was checked to catch any special cases WIVET might have. The session cookie was set by clicking the *Http Sessions* tab current, and setting the session from the browser to *active*. Finally, the WIVET folder (in the *Sites* list) was right-clicked, and the option *AJAX Spider ...* was chosen and started.

w3af

The command line version of w3af was used to scan WIVET, which is started by executing `./w3af_console`. The `web_spider` (in the crawl plugin) was enabled, and configured to target `http://target/wivet` with an `ignoreRegex` set to `.*logout.*`. This means that w3af follows all URLs that match `http://target/wivet`, except if they contain the word `logout`. To configure the scan with a session cookie, a valid cookie was first exported from Firefox, which was then added in w3af by setting the `cookie_jar_file` under `config/http-settings`. Then the scan was started.

Summary

The results of the WIVET tests are shown in [Figure 4.3](#). Arachni was able to find the most test cases of all the scanners, and both its versions found the same number of test cases with the only difference being execution speed, which is not shown in the results. Arachni v2.0dev was an order of magnitude faster than v1.0.6. Because of this, and since v2.0dev otherwise performed identically to v1.0.6, the rest of the tests are only performed with v2.0dev.

Comparing results with old test data from the benchmark mentioned earlier [\[14\]](#), two scanners performed marginally worse, and two performed significantly better:

- **Arachni** - Arachni only reached 19% in the old test data, and achieved 96% in this test. This is a significant improvement, which can be explained by improved JavaScript crawling, other development done since the benchmark was performed, and probably also an adaption to WIVET.
- **Burp Suite** - Burp Suite only reached 16% in the benchmark, and achieved 50% in this test. This is also a significant improvement. This can be explained by the same factors as for Arachni; improved JavaScript crawling, other development done since the benchmark was performed, and probably an adaption to WIVET.
- **OWASP ZAP** - OWASP ZAP reached 73% in the benchmark, and only achieved 71% in this test. This means that it performed marginally worse in this test, which could be explained by programming bugs. However, the most likely explanation of this result, is that the settings for scanning WIVET are different between the benchmark and this test.
- **w3af** - w3af reached 19% in the benchmark, and only achieved 14% in this test. This means that it performed marginally worse in this test. As in the case with OWASP ZAP, this is most likely because the settings for scanning WIVET are different between the benchmark and this test. However, w3af was the most unstable of all these scanners, which can be explained by its development model, where small changes are made frequently. This could mean that the version tested here suffered from bugs that relate to crawling.

4.3.2 WAVSEP

This section presents how each of the scanners in [Table 4.4](#) were tested on WAVSEP. A summary of the results is provided at the end of the section:

Arachni

To scan WAVSEP using Arachni, the command `./bin/arachni` was executed with the target URL, and the following options:

4.3. TESTING

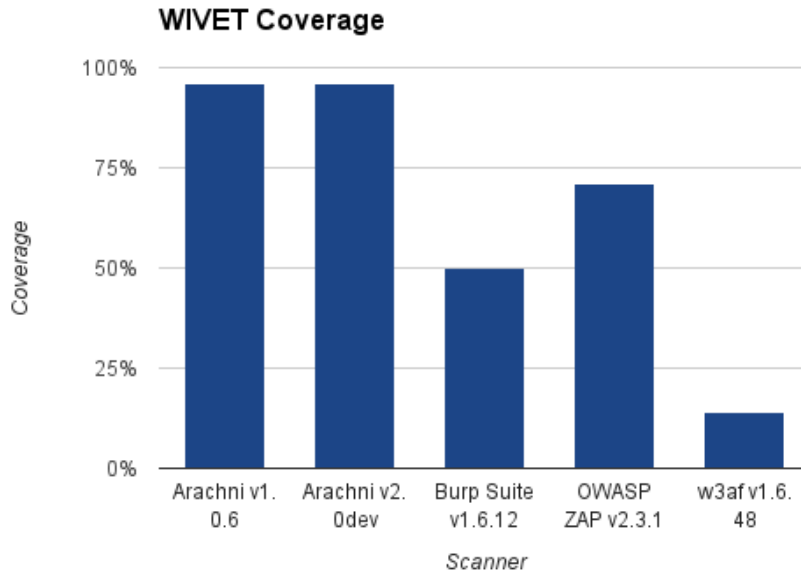


Figure 4.3. WIVET coverage results for Arachni, Burp Suite, OWASP ZAP and w3af

- **--checks** - For example `--checks=xss*` on the XSS test cases.
- **--audit-links --audit-forms** - This restricts the scanner to only audit links and forms (and not cookies, etc.). This was used to limit the scan, and thereby make it faster.

Example use when scanning *ReflectedXSS GET*:

```
./bin/arachni http://target/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/index.jsp --checks=xss* --audit-links --audit-forms
```

Table 4.5 shows the checks used for each category of test cases.

Burp Suite

To scan WAVSEP using Burp Suite, a similar approach to the one used to scan WIVET was used:

- A proxy was set up.
- The browser, using the proxy, was then used to navigate to the test URL, e.g. <http://target/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/index.jsp>.

Table 4.5. A presentation of the options used for Arachni to scan each test category

Test category	Options
Local File Inclusion/Directory Traversal/-Path Traversal	path_traversal, directory_listing, file_inclusion, source_code_disclosure
Remote File Inclusion	rfi
Reflected Cross Site Scripting (RXSS)	xss*
SQL Injection	sql*
Unvalidated Redirect	unvalidated_redirect, unvalidated_redirect_dom
Old, Backup and Unreferenced Files	backup_files, backup_directories, common_files, common_directories

- The target folder, e.g. RXSS-Detection-Evaluation-GET, was right-clicked, and the option **Spider this branch** was chosen.
- In the *Options* pane under the *Scanner* tab, the following was set:
 - Set:
 - * Scan speed: *Thorough*
 - * Scan accuracy: *Normal*
 - All the applicable options under *Active Scanning Areas* were then checked, e.g. Reflected XSS and Stored XSS.
- The target folder, e.g. RXSS-Detection-Evaluation-GET, was right-clicked, and the option **Actively scan this branch** was chosen.

The extensions *J2EEScan 1.2.3* and *Additional Scanner Checks 1.2* from the BApp Store were used to get improved coverage. [Table 4.6](#) shows the active and passive checks used for each category of test cases.

OWASP ZAP

To scan WAVSEP using OWASP ZAP the following procedure was followed:

- A proxy was set up.
- The browser, using the proxy, was then used to navigate to the test URL, e.g. <http://target/wavsep/active/Reflected-XSS/RXSS-Detection-Evaluation-GET/index.jsp>.
- The target folder, e.g. RXSS-Detection-Evaluation-GET, was right-clicked and the option **Spider Subtree** was chosen.
- The target folder, e.g. RXSS-Detection-Evaluation-GET, was right-clicked and the options **Attack** and then **Active Scan advanced ...** were selected.

4.3. TESTING

Table 4.6. A presentation of the options used for Burp Suite to scan each test category

Test category	Options
Local File Inclusion/Directory Traversal/-Path Traversal	File path traversal / manipulation, Information disclosure
Remote File Inclusion	Remote file inclusion, Information disclosure
Reflected Cross Site Scripting (RXSS)	Reflected XSS, Stored XSS, DOM XSS in Additional Scanner Checks
SQL Injection	All checks under SQL injection
Unvalidated Redirect	Open redirection
Old, Backup and Unreferenced Files	Right-click folder and choose Engagement Tools > Discover Content. Then under the Config tab check discover files and directories and enable all file extensions and start

- In the *Policy* tab, the **Default Threshold** was switched to *OFF* for all categories, and then the applicable *Policy* was configured to have **Threshold: Medium** and **Strength: Insane**, e.g. Policy=Injection > Cross Site Scripting (Reflected), Cross Site Scripting (Persistent), Cross Site Scripting (Persistent) - Prime and Cross Site Scripting (Persistent) - Spider.

The extensions *active scanner rules (beta) version 14* and *ascanrulesAlpha version 11* from the OWASP ZAP marketplace were used. The external extension *Good Old Files v1.0* [25] was also used. Table 4.7 shows the checks used to scan each category of test cases.

Table 4.7. A presentation of the options used for OWASP ZAP to scan each test category

Test category	Options
Local File Inclusion/Directory Traversal/-Path Traversal	Direct Browsing, Path Traversal
Remote File Inclusion	Remote File Inclusion
Reflected Cross Site Scripting (RXSS)	Cross Site Scripting, Cross Site Scripting (Reflected/Persistent) - Prime/Spider, Script active scan rules
SQL Injection	SQL Injection, SQL Injection - MySQL/Hypersonic SQL/Oracle/PostgreSQL
Unvalidated Redirect	External Redirect, Open Redirect
Old, Backup and Unreferenced Files	Good old files plugin

w3af

To scan WAVSEP using w3af special scripts was created, and executed with `./w3af_console -s ScriptFile.w3af`. The scripts used are a stripped version of a script found on the OWASP wiki [53]. In the version used here, the fuzz settings, the authentication settings, and the settings of target OS and framework were removed. To direct each scan different audit, grep and crawl plugins were used. Table 4.8 shows the audit, grep and crawl plugins used to scan each category of test cases.

Table 4.8. A presentation of the options used for w3af to scan each test category

Test category	Options
Local File Inclusion/Directory Traversal/-Path Traversal	lfi (audit)
Remote File Inclusion	rfi, xss (audit)
Reflected Cross Site Scripting (RXSS)	xss (audit), dom_xss (grep)
SQL Injection	sqli, blind_sql (audit)
Unvalidated Redirect	global_redirect, phishing_vector (audit)
Old, Backup and Unreferenced Files	content_negotiation, digit_sum, dir_file_bruter (bf_directories, bf_files, be_recursive), url_fuzzer, wordnet (crawl)

Summary

The results of the WAVSEP tests are divided into three separate charts:

- One showing the test coverage on each test category presented in section 3.3.1, for each of the scanners (see Figure 4.4).
- Another showing the percentage of false positives reported as vulnerable in each test category for each scanner (see Figure 4.5).
- A third showing the overall average for each scanner, in each metric (coverage and false positive) (see Figure 4.6).

Note that the absence of any bar(s) in a chart simply means that the bar(s) have the minimum value in the chart, and as such they are not shown. Full results are available in Appendix A. Coverage is measured as: $\frac{\text{number of vulnerabilities found}}{\text{total number of vulnerabilities}}$

A notable result is that Arachni is bested in the *Reflected Cross Site Scripting (RXSS)* category by OWASP ZAP, which happened since OWASP ZAP had support for more scripting languages than Arachni. Another notable result is that OWASP ZAP identified all the false positive test cases in the backup category as vulnerable (see Figure 4.5), which is most likely due to the quality of the external plugin Good Old Files [25].

4.3. TESTING

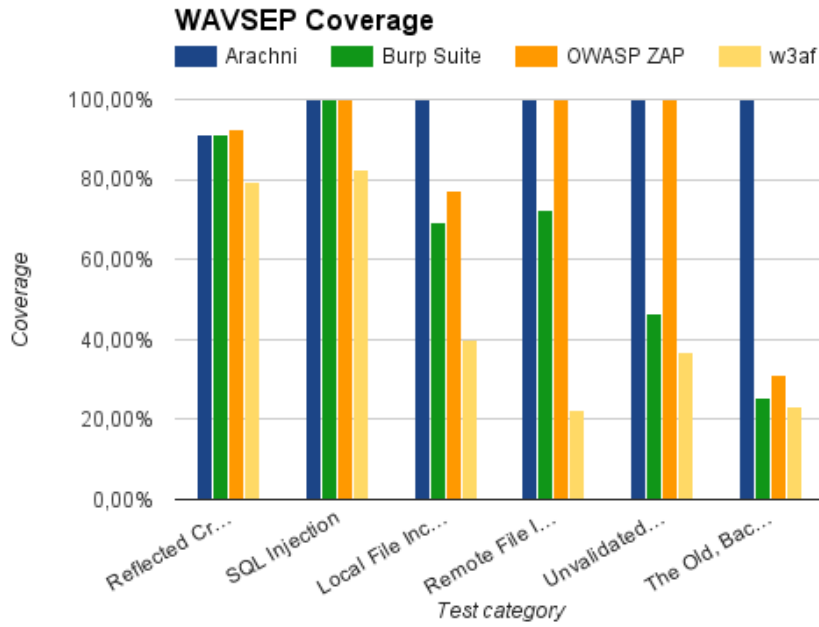


Figure 4.4. WAVSEP coverage for each test category described in section 3.3.1

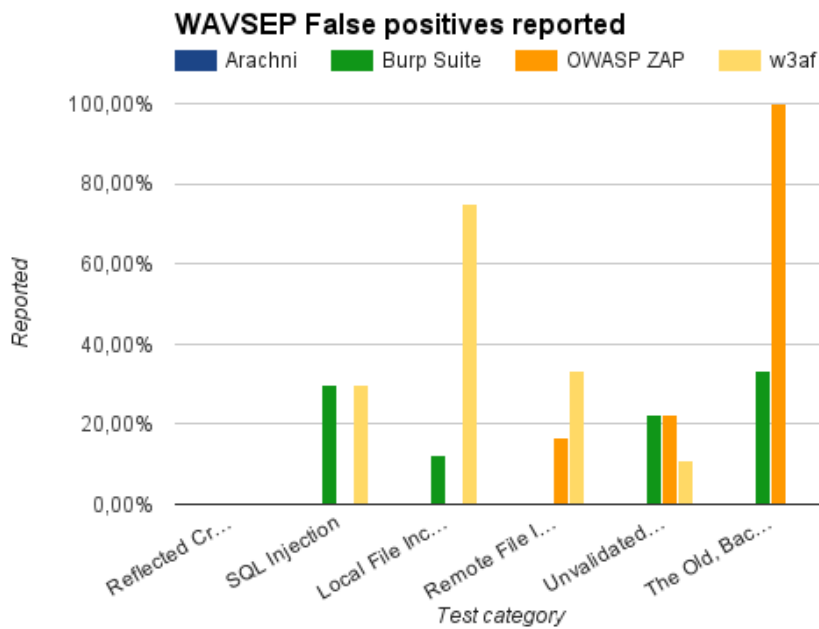


Figure 4.5. WAVSEP false positives reported for each test category described in section 3.3.1

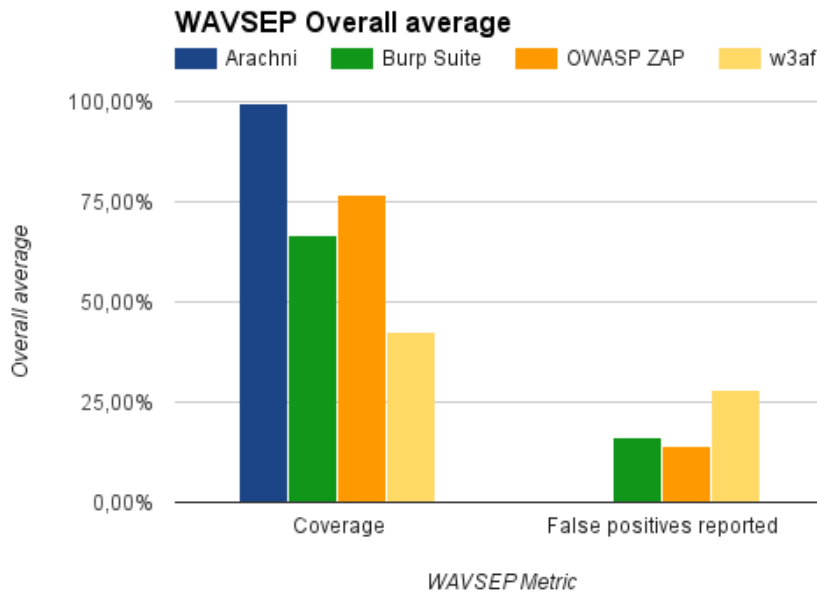


Figure 4.6. Summary of overall average WAVSEP coverage and false positives

Comparing results with old test data from the benchmark mentioned earlier [14], some more notable results were found, including some quirks and inconsistencies that do not show in the figures:

- **Arachni** - Compared to the old test data, Arachni now has support for another attack vector (Old, Backup and Unreferenced Files), it has increased coverage in all categories (that were not already 100%), and it has gotten rid of the false positives in SQL Injection. The extremely high numbers probably means that Arachni has been adapted to perform well WAVSEP

Arachni was just 6 test cases in the XSS category from reaching 100% coverage overall. The missed test cases relied on support for Flash and VBScript, which Arachni does not support, and which is not used by Nordnet.

- **Burp Suite** - Compared to the old test data, Burp Suite has increased the overall coverage on WAVSEP, while it has decreased coverage in the XSS category. The false positive rate has increased significantly, from 0% overall to roughly 16%. The cost of increasing coverage is often also an increased false positive rate, which can explain these results.

The DomXSS check, that is a part of the Additional Scanner Checks extension, did not find any of the four test cases vulnerable to DOM based XSS. Some of the test cases under SQL Injection required multiple scans to find all cases,

4.3. TESTING

hence the results were inconsistent. This could mean that network or server problems were not sufficiently handled.

- **OWASP ZAP** - Compared to the old test data, OWASP ZAP has decreased coverage in two categories (XSS and backup files), while it has increased coverage in two other categories (Unvalidated Redirect and Local File Inclusion). The false positives in the SQL Injection category have been fixed, while the false positive rate in the Unvalidated Redirect category has increased. The same observation as above can be made; increased coverage in the Unvalidated Redirect category also led to an increased false positive rate.

OWASP ZAP had even more inconsistencies than Burp Suite. These were manifested in the categories SQL Injection, Local File inclusion, Remote File Inclusion and Old, Backup and Unreferenced Files. These results are hard to explain and point to some underlying programming error since common problems, such as network errors do not normally affect static categories (file inclusion or backed up files). However, the inconsistencies in the backup category are most probably caused by the external extension that was used (*Good Old Files*).

- **w3af** - Compared to the old test data, w3af has decreased coverage in two categories (Local File Inclusion and Unvalidated Redirect), while it has increased coverage in the other categories. The false positive rate has increased dramatically overall, due to increases in both categories that deal with file inclusions (a two times higher rate in Remote File Inclusion and a six times higher rate in Local File Inclusion). The increase in false positive rate in the Local File Inclusion category stands out since it is comparatively large, and since the coverage in the same category was decreased. This could have been caused by a number of things, such as programming bugs, different settings used or it could be due to a lack of regression tests (mentioned in [section 2.3](#)).

w3af also had many inconsistencies (easiest seen in the Local File Inclusion category) and some, what appeared to be, random errors while scanning cases under Remote File Inclusion. These random errors occurred in all categories, and therefore they are probably caused by programming errors.

Overall, the comparison between the results of the benchmark and the results from the tests conducted in this thesis implies that scanners are performing both better, and worse. A possible explanation for the cases where coverage decreased between the two tests can be that different settings were used. However, this impact was minimized by using the same settings for this thesis as in the benchmark (wherever they were noted, which they were *most* of the time but not always).

After testing the scanners against WAVSEP and WIVET, w3af was removed from further testing due to its relatively poor results, and the errors and inconsistencies mentioned above. It turned out that Burp Suite and OWASP ZAP were quite similar with a slight edge to OWASP ZAP in the results (see [Figure 4.3](#) and [Figure 4.6](#)),

which it received much thanks to the highly unstable external plugin *Good Old Files*. Due to this plugin, and inconsistencies experienced during scanning, OWASP ZAP is also removed from further comparison.

4.3.3 Nordnet

Since both w3af and OWASP ZAP were removed in the last section, only Arachni and Burp Suite are tested in this section.

When this test was carried out, the existing processes for finding vulnerabilities at Nordnet (see [section 2.1](#)) had found eight XSS vulnerabilities in the test environment. Most of the details of these vulnerabilities are censored in the following test because they are sensitive to Nordnet (such as URLs and parameter names). The test results can be seen in [Figure 4.7](#). The lighter colored area behind the bars shows how many known vulnerabilities there are on each URL.

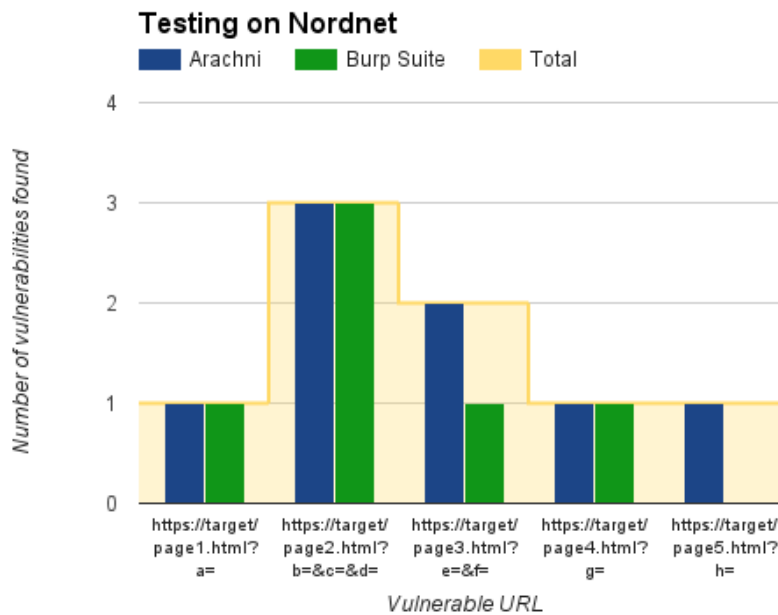


Figure 4.7. Test results for Arachni and Burp Suite on Nordnet

After this final evaluation of Arachni and Burp Suite, where Arachni once again has the highest coverage, Arachni is chosen as the scanner that will be integrated in the development process at Nordnet.

4.4 Integration and implementation

In this section the three places where security testing can be integrated in the delivery pipeline, suggested in [subsection 3.4.1](#), are evaluated. One of these is chosen, and

4.4. INTEGRATION AND IMPLEMENTATION

then Arachni is implemented there.

4.4.1 Integration

Three potential places where security testing can be integrated in the delivery pipeline are evaluated by listing their pros and cons:

- **Before committing**

- Pros:

- * The feedback from the security testing is as fast as it can possibly be.
- * The testing takes no time for the actual CI process since the testing happens outside of its scope.

- Cons:

- * The security testing is highly dependent on the security knowledge of the developer, and not fully automated.
- * There is no guarantee that the security testing has been performed when code is committed since the developer can just choose to ignore it.
- * There is no guarantee that the set up that one developer has is the same as someone else's, which can lead to inconsistencies between tests.

- **In the *CI: Automatic tests* step**

- Pros:

- * Most automated solution of these three.
- * Developers get fast feedback and QA still has a final say.
- * One central scanner is easy to maintain and ensures that everyone uses the same version and gets the same results.

- Cons:

- * The endpoints must be manually specified (the URLs that the scanner should scan, or start scanning).

- **In the *QA: Tests* step**

- Pros:

- * One central scanner is easy to maintain and ensures that everyone uses the same version and gets the same results.
- * QA has full control over what happens. They can get scan reports from Arachni after their testing is complete and from that decide whether or not to pass a build.

- * Practically invisible to the developers since they do not have to perform the testing but only react to the outcome of the tests.
- Cons:
- * Longer wait for developers to see the results from the security tests. The feedback loop is not automated in this case since deployment to the test environment can only be performed manually by QA.
 - * Much more work for QA.

The first suggestion looks good at a first glance, since the feedback for security testing is fast, and since the security testing takes no time from the CI process. However, according to the wishes and requirements from [section 4.1](#), the developers wanted the security testing to be as automated as possible, and with this approach they have to start and configure each scan manually. In short, it is too dependent on the developers, and too little on QA. There is also no guarantee that the security testing is performed once the code is committed, which means that vulnerabilities can still get out into production.

The last suggestion also looks good, since QA get full control over the scan, which they expressed a need for in the interview study. It also looks good from the developers perspective since they do not have to perform the security testing. However, this approach is too dependent on QA and that affects the feedback rate badly. Since this step must be started manually and requires active work from the QA engineers, it is not that automated either.

This leaves the second suggestion; to integrate the scanner in the *CI: Automatic tests* step. This suggestion has a good combination of both feedback rate, and involvement of QA engineers. It is also the most automated solution of the three, which satisfies the wishes from both developers and QA (see [section 4.1](#)). Finally, it means that a single scanner can be used, which makes it easy to maintain. The only issue with this suggestion is to find the endpoints for an application, but since Arachni displayed good results in the crawling tests (see [Figure 4.3](#)) this can be solved by only supplying it with a few starting points, and then letting Arachni crawl through the rest of the application. Therefore, for this approach to work, it was decided that each project must supply a file containing endpoints in JSON [20] format. An example of such a file is shown below:

Listing 4.1. An example endpoints file

```
{
  "endpoints": [
    "http://target.ci/endpoint1",
    "http://target.ci/endpoint2",
    "http://target.ci/endpoint3"
  ]
}
```

4.4. INTEGRATION AND IMPLEMENTATION

4.4.2 Implementation

To implement Arachni in the *CI: Automatic tests* step, a small wrapper script [12] was written, and a *scan profile* targeting Nordnet web applications with all applicable checks from subsection 4.2.1 was created. The wrapper script is designed to be called with the endpoints file as an argument, and it has the following tasks:

- Check the environment
 - Is the scanner located where it should be?
 - Is the scanner profile located where it should be?
- Parse data and construct a call to Arachni
 - Parse the endpoints file for URLs and convert them to a form that Arachni understands, e.g. the contents of Listing 4.1 would be converted to:
`"http://target.ci" --scope-extend-paths=EXTEND_PATHS.`
EXTEND_PATHS is a file created by the script that contains one row per endpoint (/endpoint1, /endpoint2 and /endpoint3).
 - Generate a unique name for the scan.
 - Call Arachni with the arguments above.
- Generate a scan report in JSON format from the results of the scan with the help of `arachni_reporter`, which is part of Arachni [33].
- Choose parts of the JSON formatted scan report to build a JUnit [30] formatted .xml scan report that Jenkins [31] can understand.

To meet the requirements posed in subsection 3.4.2, and to adapt the report to the security knowledge of the target group, the following items are included in a scan report:

- The category for the vulnerability as class name in the JUnit report. Since JUnit is built for Java [42] and collects all issues per class this means that all issues in the same category are grouped together in the published report.
- **URL** - This is used as both issue name in the scan report and as a field in the description. In the scan report all issues are named `<classname>.<name>`. Having named the issues like this makes it easier to distinguish between issues.
- For each issue there is a description (mentioned in the above item), which includes the following:
 - **Title** - A title for the issue, a longer version of the category above, e.g. an issue with the category `XSS_TAG` receives the title *Cross-Site Scripting (XSS) in HTML tag*.

- **Severity** - The severity of the issue, this is any one of the following (sorted from lower to higher): informational, low, medium, high.
- **Description** - A short description of the issue.
- **References** - Some references to the issue from well-known sources such as OWASP [46].
- **Variations** - This is a list of all variations found by the scanner of how to exploit the specific vulnerability. The following is included for each variation:
 - * **Trust** - If the variation is trusted or not (if it is untrusted it must be verified manually).
 - * **Affected page** - A full URL of the exploit, works best when Method (below) is GET.
 - * If they are available, these are also included:
 - **Injected** - The injected payload.
 - **Proof** - Proof that the exploit worked.
 - **HTTP request** - The full HTTP request with all headers. This is a good complement to the affected page since this takes care of all HTTP methods (POST, PUT, etc.).
- The description also include the following items, if they are available:
 - * **Method** - The HTTP method used by the scanner when it found the issue (e.g. GET, POST etc.)
 - * **Parameter** - The vulnerable parameter in this specific case.
 - * **Remediation guidance** - Some general information on how to remedy the vulnerability.

When such a scan report is created, Jenkins can read and publish it. Based on the scan report the script also decides what to do with the build: If any issues are found during a scan, the script marks the build as unstable, which can be seen in [Figure 4.8](#). When a step in the delivery pipeline (see [Figure 2.4](#)) is marked as unstable this is an indication that something may be wrong, but it is still possible for a QA engineer to manually allow the build to continue. Marking a build unstable also triggers a mail to the affected developers to notify them of the report. Jenkins also keeps track of the issues over several builds, which means that it is possible to see how old an issue is (for how many builds it has been around), and if the number of issues reported are more, less or the same as in previous builds (see [Figure 4.9](#)). An example of a scan report can be seen in [Figure 4.10](#).

4.5 Guide

When working with the last section, it made more sense to have any remediation advice information directly available in Jenkins [31], rather than in the guide as

4.5. GUIDE

Build Pipeline

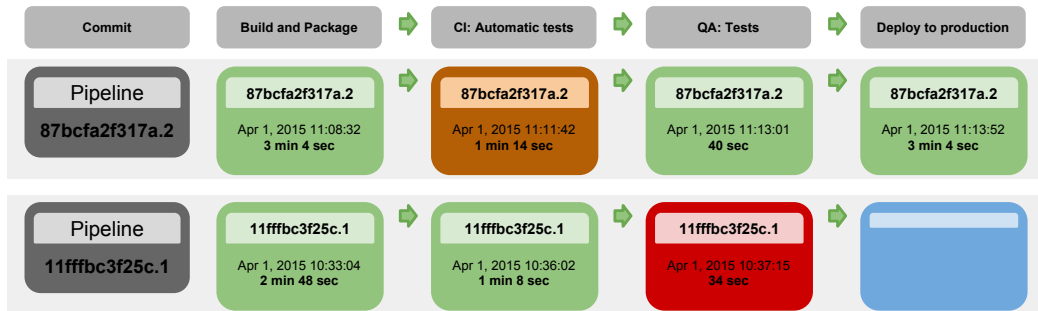


Figure 4.8. A version of [Figure 2.4](#) where the *CI: Automatic tests* step is marked as unstable by the automated security tests

Test Result

3 failures (+1)

7 tests (+2)

[Took 1 min 32 sec.](#)

All Failed Tests

Test Name	Duration	Age
+ XSS.http://target.ci/endpoint1/	24 sec	1
+ XSS.http://target.ci/endpoint2/	14 sec	1
+ XSS_TAG.http://target.ci/endpoint3/	54 sec	1

All Tests

Package	Duration	Fail	Skip	Pass	Total
(root)	1 min 32 sec	3	0	4	7

Figure 4.9. A summarized view of the report in Jenkins [\[31\]](#). This view is accessed by clicking on the unstable step in [Figure 4.8](#) and then on *Test Result*

```

Failed

XSS_TAG.http://target.ci/endpoint1/

Error Message
XSS in HTML tag

Stacktrace
Cross-Site Scripting (XSS) in HTML tag - XSS in HTML tag
-----

* Severity: HIGH
* URL:      http://target.ci/endpoint1/
* Method:   GET
* Parameter: param

#####
# Variations #
#####

Variation 1 (Trusted)
* Injected: 1" arachni_xss_in_tag="374d8e30118449a9946323f960852279" blah="
* Proof:    " arachni_xss_in_tag="374d8e30118449a9946323f960852279" blah="

* Affected page: http://target.ci/endpoint1/?param=1%22%20arachni_xss_in_tag=%
22374d8e30118449a9946323f960852279%22%20blah=%22

* HTTP request:
GET /endpoint1/?param=1%22%20arachni_xss_in_tag%3D%
22374d8e30118449a9946323f9608522
79%22%20blah%3D%22 HTTP/1.1
Host: target.ci
Accept-Encoding: gzip, deflate
User-Agent: ci_scanner
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

#####
# Description #
#####

Client-side scripts are used extensively by modern web applications.
They perform from simple functions (such as the formatting of text) up to full
manipulation of client-side data and Operating System interaction

```

Figure 4.10. A sample scan report in Jenkins [31]. See subsection 3.4.2 for all the information that this report includes

4.5. GUIDE

suggested in [section 3.5](#). Therefore the guide contains some general information about how to set up security testing on a project, how to interpret scan reports, and some examples to make the work easier to grasp for the readers (basically other versions of [Figure 4.9](#), [Figure 4.8](#) and [Figure 4.10](#)).

When looking through internal guides at Nordnet, there was already a guide that included specific coding advice called *Secure Coding Guideline*. Therefore, that guide is referred to when necessary to avoid creating duplicate content.

The final guide is named *Security Testing in the CI Environment*, and can be found in [Appendix B](#). The contents of the guide are:

- **Introduction** - This section introduces the guide; who it is for and what it covers.
- **Setup** - This section explains how to modify a project so that it can be tested in the *CI: Automatic tests* step.
- **Report** - This section explains what is included in the scan report and why. This is the same information that is available in [subsection 4.4.2](#).
- **Examples** - This section contains examples of what it looks like when everything is set up correctly. The figures in this section are roughly the same as the ones in [subsection 4.4.2](#).

Chapter 5

Conclusions

In this chapter the results of this thesis will be discussed and conclusions will be given. Recommendations and future work will also be addressed.

5.1 Discussion

All the areas that are defined in the purpose (see [section 1.3](#)) are covered in the thesis. First the goals stated in the purpose will be discussed, and then the results from [chapter 4](#), and lastly an answer to the problem statement (in [section 1.2](#)) will be discussed.

5.1.1 Goals

Here are the goals stated in the purpose (see [section 1.3](#)):

- **Investigate how security can be integrated in an Agile development process**

This was investigated by first looking at any theory related to this question. From that, two plausible suggestions of how to integrate security in the development process were found: *source code analysis* and *automatic security testing* using a scanner (see [Figure 2.7](#)) [11, 32]. Both of these are automated once set up, which fits the goal of Agile perfectly. I chose to go with the automatic security testing approach since it has the major advantage of being independent of how an application is built, whereas code analysis is highly dependent on what programming language is used (see [subsection 3.1.1](#)). This makes the automatic security testing approach more general and easier to adapt than static analysis, but it also brings some limitations. The cost of using a scanner is that it can miss whole classes of vulnerabilities [17, 13]. However, more recent data shows that scanners are getting better at finding vulnerabilities [14], and under the assumption that the scanner supports the most common web application vulnerabilities (see [subsection 2.4.1](#)) an automatic security testing

approach using a scanner is the best way of integrating security in an Agile development process.

- **Investigate how and where automated security tests can be included in continuous integration within the Agile development process**

To investigate *how* this could be done, a pre study was conducted (see [section 4.1](#)). This contained an evaluation of the two suggestions mentioned above, and an interview study that was used to get requirements and wishes from Nordnet. Since the automatic security testing approach was chosen above, security will be integrated in the Agile development process by using a web application security scanner. To investigate *where* in the development process the scanner could be integrated, theory about CI [24] and the development process at Nordnet was studied. This turned out into a model development process shown in [Figure 2.4](#). An interview study was also conducted in order to get ideas and requirements for the scanners placement in the development process. Three places in the delivery pipeline where security tests could be included were identified (see [Figure 3.1](#)). The pros and cons for each of the three places were analyzed, and finally the *CI: Automatic tests* step of [Figure 2.4](#) was chosen since it provided the most automated testing. It also enabled fast feedback for developers, while still letting QA have full control over the build (see [subsection 4.4.1](#)).

- **Produce a guide for developers telling them how to set up automated security tests, helping them to understand security implications and risks and showing them how they can mitigate certain risks to reduce vulnerabilities**

This was approached by using internal Nordnet guides, and OWASP testing guides as inspiration in order to create a guide that would feel familiar to the future readers at Nordnet, and to make sure that the contents were properly adapted to development and testing. The contents of the guide are highly influenced by the outcome of the goals above and of requirements and wishes from Nordnet collected in the interview study (see [section 4.1](#)). These interviews were also used to determine the general level of security knowledge on Nordnet, which in turn was used as an indication of what level the guide should be written on.

The guide ended up containing information about how to set up security testing for a project, what a scan report contains, and some examples of how it will look like once set up (see [section 4.5](#)). Since security implications, risks and mitigation tips are all highly coupled with the issue type I decided to put those in a the scan report instead of in the guide as originally planned. This scan report contains information about all issues found during a scan and is published to Jenkins [31] (see [Figure 4.10](#)). For the final guide see [Appendix B](#).

5.1. DISCUSSION

5.1.2 Results

In this subsection the results found in [chapter 4](#) will be discussed.

Pre study

This is a special section since it contains requirements for all the work performed in this thesis (see [section 4.1](#)). Therefore I will explain either how each requirement is met, or motivate if it is not:

- **General:**

- The scanner can be interacted with from the command line:
This is met since Arachni has a rich CLI (see the documentation [33], and the wrapper script [12]).
- The scanner should be *actively* maintained:
Active is not an absolute measure. The following criteria were used to determine if a scanner was actively maintained: time since last stable release, time since last stable commit and who uses it. For Arachni, the last stable release when the testing was performed (March 2015) was on the 8th of December 2014, and during testing a development version was used, which is now (May 2015) released. There are currently testimonials from ten companies using Arachni (see the liaison program at the Arachni website) with the most notable being eBay [19]. [33]
- The false positive ratio should be *as low as possible*:
This is one of the mandatory features required in a web application security scanner by NIST [9]. Arachni has the lowest false positive ratio [48] of all the tested scanners in each of the test categories of WAVSEP (see [Figure 4.5](#)) and also overall (see [Figure 4.6](#)).
- The false negative ratio should be *as low as possible*:
Arachni has the highest coverage (number of exploits found) of all the tested scanners, both on WAVSEP (see [Figure 4.4](#)) and on Nordnet (see [Figure 4.7](#)). A high coverage implies a low false negative ratio [47].
- The scanner can handle basic authentication (can scan as a logged in user):
Arachni can handle authenticated scans, supporting both Kerberos and HTTP authentication (see Arachni documentation [33]).
- Reports should gather similar issues to avoid ambiguities:
By using the vulnerability category as class name in the scan reports similar issues were grouped together, see [Figure 4.9](#).

- **From developers:**

- Any reports should be easily understood for developers even if they are security novices:

I tried to meet this requirement by showing early versions of the scan report (see [Figure 4.10](#)) to developers and then modifying it according to their feedback. A more extensive evaluation of this requirement is outside the scope of this thesis.

- Developers should not have to write any extensive tests (scanning should be as automated as possible):

Developers do not need to write any tests at all since the actual scan is fully automated, but it requires a minor set up (see [subsection 4.4.1](#)). See the guide in [Appendix B](#) for more information about this.

- **From people responsible for the CI environment / Jenkins:**

- The scan needs to be fast, maximum five minutes:

This requirement is important since one of the core principles of the Agile method is speed (see [section 2.2](#)). It was met by supplying a timeout of 5 minutes to Arachni (see the script at [GitHub \[12\]](#) and Arachni documentation [[33](#)]). Having a fixed timeout is not perfect, since it could lead to unnoticed issues, and if the scan is deterministic (follows the same pattern every time) this can lead to some issues never being detected. The 5 minute timeout was never hit during testing, but in the future it may be needed to configure the scan settings differently to make it faster, maybe by dividing the scan on several machines or by limiting the checks made.

- The scanner should preferably be able to run on Linux:

Arachni runs on Linux (see Arachni documentation [[33](#)]).

- The scanner should indicate how the scan went by exiting with certain codes or by providing a scan report in JUnit [[30](#)] format:

This is solved by the wrapper script. It will read the scan report after the scan and provide an appropriate exit code and/or a JUnit report, see the script at [GitHub \[12\]](#) and [Figure 4.10](#).

- **From QA:**

- The ability to somehow mark false positives and hide those in future scans:

This is possible to do, but implementing it is outside the scope of this thesis. Since each issue is assigned a unique id (hash) (see the documentation of Arachni [[33](#)]), which could be saved in some sort of database it would be possible to track the same issue appearing over several scans, and thereby also ignore it.

5.1. DISCUSSION

- The automated scan should not be able to fail the build on its own:
The automated scan marks a step as unstable if any issues are found; It does not fail the entire build. An example of this can be seen in [Figure 4.8](#), where the CI: Automatic tests step is marked as unstable in the upper row, and the project is still deployed to production.
- The ability to manually mark the build as *pass* or *fail* based on the scan report:
Based on the scan report (see [Figure 4.10](#)) the QA can mark the QA: Tests step as pass or fail, and if they mark it as fail the whole build will be marked as fail as well. See [Figure 4.8](#) for an example where the QA: Tests step is marked as fail in the bottom row, and hence so is the whole build.

Evaluating scanners

The evaluation of scanners was performed by translating the technology at Nordnet, and the requirements gathered in the interview study, into a list of applicable attack vectors that a scanner should support. Based on those interviews and a web application security scanner specification from NIST [9], the attack vectors that were not applicable were motivated and removed, and the rest were marked as applicable, or unknown (if something was unclear) (see [Table 4.1](#)).

In [Table 4.2](#) 20 different scanners are mapped to these attack vectors. The four scanners that were chosen for testing (see [Table 4.3](#)) were not the best in terms of *number of supported attack vectors*, instead of looking at that number while comparing the scanners, several criteria was defined (see [subsection 3.2.2](#)). Among others these criteria contained looking at groups of attack vectors together, for example: columns C, D and E are all targeting XSS (see [Table 4.1](#)), and since Wapiti supports two of three (C and D) and Jsky only one of three (C), Wapiti is preferable over Jsky, even though Jsky supports one more attack vector.

Testing

The results of the WIVET coverage test shown in [Figure 4.3](#) and the overall summary of the WAVSEP test shown in [Figure 4.6](#) clearly shows that Arachni [33] is the most suitable scanner in this context. Arachni did not reach 100% coverage since it does not support VBScript or Flash, but since these are not used at Nordnet it does not matter.

To verify the results of the tests on WIVET and WAVSEP, Arachni and Burp Suite were also tested at applications provided by Nordnet. This also helps avoiding that the results are biased by deliberately vulnerable applications (see [subsection 3.3.1](#)). The two scanners ended up being tested on eight XSS vulnerabilities (see [Figure 4.7](#)). Arachni found all the vulnerabilities and therefore it was chosen for the integration.

Integration and implementation

Three potential places to integrate security testing in the Agile development process are compared, and the most automated of these suggestions end up being chosen, which is to integrate security testing in the *CI: Automatic tests* step of [Figure 3.1](#). Arachni is then implemented there by using a wrapper script [12]. The finished delivery pipeline can be seen in [Figure 4.8](#) and it supports *nearly* all requirements gathered in the interview study, as motivated above. Once a scan is complete a scan report will be published to Jenkins, which can be accessed in a summarized form (see [Figure 4.9](#), or as a full report (see [Figure 4.10](#)). The report contain mandatory and optional items as described by NIST in [subsection 3.4.2](#), which includes information such as: the attack that was made, a proof that the attack succeeded, and tips on how to mitigate the issue. Additional information is also included in the report to make sure that everyone in the target group of this thesis can understand the issue, and to help developers and QA to replicate, verify and fix it.

Guide

The guide in [Appendix B](#) was influenced by internal guidelines found at Nordnet and by two OWASP guides [44, 43], to make sure that it would feel familiar to the readers and that the contents matched their level of security knowledge. The final guide contains a short introduction that presents the intent of the guide, a section explaining how to set up security testing in a project, a section explaining the different parts of a scan report, and a section containing examples of what everything will look like once set up (see [section 4.5](#)).

5.2 Integrating Automated Security Testing in the Agile Development Process

Traditionally, security has been separated from the Agile development process, leading to a development process that is neither truly Agile nor fully secure [11]. This has led to the release of insecure software, which in turn has led to the big number of vulnerabilities discovered during the last few years (see [Figure 2.6](#)). I have tried to make the Agile development process more secure by investigating the following question (asked in the problem statement, see [section 1.2](#)):

“How can automated security tests be included in the Agile development process in order to avoid vulnerabilities in production?”

Since this thesis focuses on the development of web applications, I have attempted to solve the problem by investigating how a web application security scanner can be integrated in the CI delivery pipeline (see [subsection 2.2.1](#) and [Figure 4.8](#)). This means that the application will be scanned each time it builds (on every commit). After each scan, a scan report containing information about any issues discovered will be published to Jenkins (see [Figure 4.9](#) and [Figure 4.10](#)). This report is intended

5.3. RECOMMENDATIONS

for both developers, by providing information about the issues so they can start fixing them, and for QA engineers, by providing information so that they are able to verify each issue (see [subsection 4.4.2](#)), and therefore get enough information to judge whether to *pass* or *fail* a build. Since a scan report is available early in the delivery pipeline, developers will get fast feedback, and therefore they will know about, and be able to fix any security related issues reported by the scanner, before these reach production. There is a great deal of money that can be saved by employing this approach, since the cost of fixing a bug is heavily related to where it is found (see [Figure 2.5](#)). This means that the development process also becomes more effective by using this approach.

This type of security testing is very easy to set up on a new or existing project, and it is relatively easy to get started with since it is aimed at people who are security novices. Therefore, the approach described in this thesis makes the Agile development process more secure in a simple way. Since the integration and implementation is made simple (see [section 4.4](#) and [section 4.5](#)), the risk that the suggested approach is going to be thrown away or sidestepped because of complexity is very low. The suggested approach in this thesis is adapted to Nordnet, but it should also be general enough to fit any Agile development process with minor tweaking. The delivery pipeline and all of the technical requirements in [section 4.1](#) are for example all specific to Nordnet, but by following the same approach as this thesis it is possible to come up with something similar in any Agile development process.

Since there is no security in the Agile development process originally (see [section 1.2](#) and [section 2.2](#)), making it secure may not seem like the hardest thing to do. What I suggest does not *only* mean that the Agile development process gets **secure**, but also more **effective**. However, this does not mean that what I suggest makes the process *fully* secure; there is still much left to do. Some suggestions for future work will be included later.

5.3 Recommendations

As discussed above, the solution that I propose is not perfect, but it is undoubtedly a big step in the right direction. Therefore I advise anyone who wants a more secure Agile development work flow to incorporate security testing by integrating a vulnerability scanner (see [subsection 2.5.2](#)) in the testing step of the CI delivery pipeline (see [subsection 2.2.1](#)). This is an easy to implement (see [section 4.4](#)) and easy to use (see [section 4.5](#)) first step to achieving a more secure and effective Agile development process.

This approach can later be extended, and in the next section some ways of doing that are suggested.

5.4 Future work

Due to the increase of vulnerabilities in the IT sector over the last few years (see [Figure 2.5](#)), the importance of IT security is growing. At the same time as this is happening, software development is moving towards an Agile approach (see [Figure 2.3](#)), where security is not even mentioned (see [section 1.2](#)). Research about a secure Agile development process is just starting to kick off with the first ever workshop on Agile Secure Software Development [3] being held this August (2015).

I hope that this thesis can be a useful resource for future works. Some possible extensions that I have thought about are:

- Extend the work of this thesis. Some ideas are:
 - Investigate and evaluate what I suggest in this thesis. Does the scan report contain enough information for the developer, and can it be understood by a security novice?
 - Investigate how to mark and remember false positives to avoid reporting them over and over.
 - Investigate if endpoint detection could be done automatically, without the endpoints.json file.
 - Investigate the possibility to do something similar for other software.
- Investigate methods of integrating security elsewhere in the process; maybe in the user stories as described in [32], or by using secure code libraries.
- Compare the suggestions I have made in this thesis to other ways of integrating security in the Agile development process, perhaps to static code analysis.
- Investigate if using both black-box and white-box testing together can find even more vulnerabilities.

Bibliography

- [1] Acunetix. *Acunetix Web Vulnerability Scanner*. 2015. URL: <http://www.acunetix.com/vulnerability-scanner/> (visited on 05/07/2015).
- [2] Agile Alliance. *What is Agile Software Development?* 2015. URL: <http://www.agilealliance.org/the-alliance/what-is-agile/> (visited on 05/07/2015).
- [3] ASSD. *The First International Workshop on Agile Secure Software Development*. 24–28 Aug, 2015. Toulouse, France. URL: <http://www.ares-conference.eu/conference/workshops/assd-2015/>.
- [4] M. A. Awad. “A Comparison between Agile and Traditional Software Development Methodologies”. Honors Thesis. University of Western Australia, 2005.
- [5] K. Louise Barriball and Alison While. “Collecting data using a semi-structured interview: a discussion paper”. In: *Journal of Advanced Nursing* 19 (1994), pp. 328–335. DOI: [10.1111/j.1365-2648.1994.tb01088.x](https://doi.org/10.1111/j.1365-2648.1994.tb01088.x).
- [6] Kent Beck. *Test Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.
- [7] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: <http://www.agilemanifesto.org/> (visited on 05/07/2015).
- [8] Simon Bennetts. *OWASP Zed Attack Proxy Project - OWASP*. 2015. URL: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project (visited on 05/07/2015).
- [9] Paul E. Black et al. “Software Assurance Tools: Web Application Security Scanner Functional Specification Version 1.0”. In: *National Institute of Standards and Technology* (2008).
- [10] Grady Booch. *Object Oriented Design with Applications*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991. ISBN: 0805300910.
- [11] Helen Bravo. “DevOps and Security: It’s Happening. Right Now.” Front Range OWASP Conference 2013. 2013. URL: https://www.owasp.org/index.php/Front_Range_OWASP_Conference_2013/Sessions/Sess4_Tech1.
- [12] Andreas Broström. *CI scanner*. 2015. URL: <https://gist.github.com/abrostrom/74859c3e863c3121ba96> (visited on 06/05/2015).

BIBLIOGRAPHY

- [13] Shay Chen. *Commercial Web Application Scanner Benchmark*. 2011. URL: <http://sectooladdict.blogspot.se/2011/08/commercial-web-application-scanner.html> (visited on 05/07/2015).
- [14] Shay Chen. *WAVSEP Web Application Scanner Benchmark 2014*. 2014. URL: <http://sectooladdict.blogspot.se/2014/02/wavsep-web-application-scanner.html> (visited on 05/07/2015).
- [15] Shay Chen. *WAVSEP - Web Application Vulnerability Scanner Evaluation Project*. 2014. URL: <https://code.google.com/p/wavsep/> (visited on 05/07/2015).
- [16] R. E. Davis et al. “Interviewer effects in public health surveys”. In: *Health Education Research* 25.1 (2010), pp. 14–26. DOI: [10.1093/her/cyp046](https://doi.org/10.1093/her/cyp046).
- [17] Adam Doupé, Marco Cova, and Giovanni Vigna. “Why Johnny Can’t Pentest: An Analysis of Black-box Web Vulnerability Scanners”. In: *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA’10. Bonn, Germany: Springer-Verlag, 2010, pp. 111–131. ISBN: 3642142141, 9783642142147.
- [18] Tore Dybå and Torgeir Dingsøy. “Empirical studies of agile software development: A systematic review”. In: *Information and Software Technology* 50.9–10 (2008), pp. 833–859. ISSN: 0950-5849. DOI: <http://dx.doi.org/10.1016/j.infsof.2008.01.006>.
- [19] eBay Inc. *Electronics, Cars, Fashion, Collectibles, Coupons and More Online Shopping | eBay*. 2015. URL: <http://www.ebay.com/> (visited on 05/12/2015).
- [20] Ecma International. *JSON*. 2013. URL: <http://www.json.org/> (visited on 05/07/2015).
- [21] Joseph Feiman and Neil MacDonald. *Magic Quadrant for Application Security Testing*. Tech. rep. 2014. URL: <https://www.gartner.com/doc/2786417>.
- [22] Finansinspektionen. *Finansinspektionen*. 2015. URL: <http://www.fi.se> (visited on 06/03/2015).
- [23] Forrester Research. *The Software Security Risk Report*. Tech. rep. 2012. URL: <http://www.coverity.com/library/pdf/the-software-security-risk-report.pdf>.
- [24] Martin Fowler. *Continuous Integration*. 2006. URL: <http://martinfowler.com/articles/continuousIntegration.html> (visited on 05/07/2015).
- [25] Michal Goldstein. *Good Old Files*. 2013. URL: <https://github.com/hacktics/good-old-files> (visited on 05/07/2015).
- [26] G.K. Hanssen, D. Smite, and N.B. Moe. “Signs of Agile Trends in Global Software Engineering Research: A Tertiary Study”. In: *Global Software Engineering Workshop (ICGSEW), 2011 Sixth IEEE International Conference on*. Aug. 2011, pp. 17–23. DOI: [10.1109/ICGSE-W.2011.12](https://doi.org/10.1109/ICGSE-W.2011.12).

BIBLIOGRAPHY

- [27] Margaret C. Harrell and Melissa A. Bradley. *Data Collection Methods: Semi-Structured Interviews and Focus Groups*. Tech. rep. Santa Monica, CA: RAND Corporation, 2009. URL: http://www.rand.org/pubs/technical_reports/TR718.
- [28] Yao-Wen Huang et al. “Securing Web Application Code by Static Analysis and Runtime Protection”. In: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. New York, NY, USA: ACM, 2004, pp. 40–52. ISBN: 1-58113-844-X. DOI: [10.1145/988672.988679](https://doi.org/10.1145/988672.988679).
- [29] Jason Huggins, Shinya Katasani, and Patrick Lightbody. *Selenium - Web Browser Automation*. 2015. URL: <http://www.seleniumhq.org/> (visited on 05/07/2015).
- [30] JUnit. *JUnit - About*. 2014. URL: <http://junit.org/> (visited on 05/07/2015).
- [31] Kohsuke Kawaguchi, R.Tyler Croy, and Andrew Bayer. *Welcome to Jenkins CI! / Jenkins CI*. 2015. URL: <https://jenkins-ci.org/> (visited on 05/07/2015).
- [32] Vidar Kongsli. “Towards Agile Security in Web Applications”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 805–808. ISBN: 159593491X. DOI: [10.1145/1176617.1176727](https://doi.org/10.1145/1176617.1176727).
- [33] Tasos Laskos. *Arachni - Web Application Security Scanner Framework*. 2015. URL: <http://www.arachni-scanner.com/> (visited on 05/07/2015).
- [34] Aditya P. Mathur. *Foundations of Software Testing*. 2nd ed. Pearson India, 2013. ISBN: 8131794768, 97881317947601.
- [35] Bruce Mayhew. *OWASP WebGoat Project*. 2015. URL: https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project (visited on 05/07/2015).
- [36] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN: 0321356705, 9780321356703.
- [37] MITRE Corporation. *CWE-928: Weaknesses in OWASP Top Ten*. 2013. URL: <https://cwe.mitre.org/data/definitions/928.html> (visited on 05/07/2015).
- [38] NIST. *Statistics Results*. 2007. URL: <https://nvd.nist.gov/cvss.cfm> (visited on 05/07/2015).
- [39] NIST. *Statistics Results*. 2015. URL: https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&pub_date_start_month=0&pub_date_start_year=2000&pub_date_end_month=11&pub_date_end_year=2014 (visited on 05/07/2015).
- [40] Nordnet AB. *IT security - nordnet corporate web*. 2015. URL: <http://org.nordnet.se/corporate-governance/internal-control/it-security> (visited on 05/07/2015).
- [41] Nordnet AB. *nordnet corporate web*. 2015. URL: <http://org.nordnet.se/> (visited on 05/07/2015).

BIBLIOGRAPHY

- [42] Oracle. *java.com: Java + You*. 2015. URL: <https://www.java.com/en/> (visited on 05/15/2015).
- [43] OWASP. *OWASP Guide Project*. Tech. rep. 2.0.1. 2005. URL: https://www.owasp.org/index.php/OWASP_Guide_Project.
- [44] OWASP. *OWASP Testing Project*. Tech. rep. 4.0. 2013. URL: https://www.owasp.org/index.php/OWASP_Testing_Project.
- [45] OWASP. *OWASP Top Ten Project*. 2015. URL: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project (visited on 05/07/2015).
- [46] OWASP. *The Open Web Application Security Project*. 2015. URL: https://www.owasp.org/index.php/Main_Page (visited on 05/07/2015).
- [47] Daniel Owen. *SANS: What is a false negative?* 2015. URL: https://www.sans.org/security-resources/idfaq/false_negative.php (visited on 05/07/2015).
- [48] Daniel Owen. *SANS: What is a false positive and why are false positives a problem?* 2015. URL: https://www.sans.org/security-resources/idfaq/false_positive.php (visited on 05/07/2015).
- [49] Ron Patton. *Software Testing*. 2nd ed. Indianapolis, IN, USA: Sams, 2005. ISBN: 0672327988, 9780672327988.
- [50] PMD. *PMD - Don't shoot the messenger*. 2015. URL: <https://github.com/pmd/pmd> (visited on 06/03/2015).
- [51] PortSwigger Ltd. *Burp Suite*. 2015. URL: <http://portswigger.net/burp/> (visited on 05/07/2015).
- [52] Andres Riancho. *w3af - Open Source Web Application Security Scanner*. 2015. URL: <http://w3af.org/> (visited on 05/07/2015).
- [53] Dominique Righetto. *Automated Audit using W3AF*. 2013. URL: https://www.owasp.org/index.php/Automated_Audit_using_W3AF (visited on 05/07/2015).
- [54] W. W. Royce. “Managing the Development of Large Software Systems: Concepts and Techniques”. In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0897912160.
- [55] Juha Sääskilahti and Juha Röning. “Challenges with Software Security in Agile Software Development”. Internal Ericsson Documentation. 2011.
- [56] Pivotal Software. *Spring Framework Reference Documentation*. 2014. URL: <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/> (visited on 06/04/2015).
- [57] Eugene H. Spafford. “The Internet Worm Program: An Analysis”. In: *SIGCOMM Comput. Commun. Rev.* 19.1 (Jan. 1989), pp. 17–57. ISSN: 0146-4833. DOI: 10.1145/66093.66095.

BIBLIOGRAPHY

- [58] Standish Group International. *Chaos Manifesto*. Tech. rep. 2011.
- [59] A. Tappenden et al. “Agile security testing of Web-based systems via HTTPUnit”. In: *Agile Conference, 2005. Proceedings*. July 2005, pp. 29–38. DOI: [10.1109/ADC.2005.11](https://doi.org/10.1109/ADC.2005.11).
- [60] UBM Tech. *Black Hat | Europe 2013 - Speakers, Shay Chen*. 2013. URL: <https://www.blackhat.com/eu-13/speakers/Shay-Chen.html> (visited on 05/07/2015).
- [61] University of Maryland. *FindBugs*. 2015. URL: <http://findbugs.sourceforge.net/> (visited on 06/03/2015).
- [62] Bedirhan Urgun. *WIVET - Web Input Vector Extractor Teaser*. 2014. URL: <https://github.com/bedirhan/wivet> (visited on 05/07/2015).
- [63] Veracode. *The Software Security Risk Report*. Tech. rep. 5. 2013. URL: <https://www.veracode.com/sites/default/files/Resources/Reports/state-of-software-security-volume-5-report.pdf>.
- [64] Stephen de Vries. *ContinuumSecurity*. 2015. URL: <http://www.continuumsecurity.net/> (visited on 05/07/2015).
- [65] Chwan-Hwa (John) Wu and J. David Irwin. *Introduction to Computer Networks and Cybersecurity*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2013. ISBN: 1466572132, 9781466572133.

Appendix A

WAVSEP results

A.1 Arachni

WAVSEP v1.5	Arachni v2.0dev			
<u>Categories</u>	<u>Response Type</u>	<u>Method</u>	<u>Total</u>	<u>Detect</u>
Reflected Cross Site Scripting (RXSS)	ReflectedXSS	HTTP GET	32	29
<i>Coverage = 91,18%</i>	ReflectedXSS	HTTP POST	32	29
<i>False positives reported = 0,00%</i>	DomXSS	HTTP GET	4	4
	<i>False positive test cases</i>	HTTP GET	7	0
SQL Injection	Erroneous 500 Responses	HTTP GET	20	20
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	20	20
<i>False positives reported = 0,00%</i>	Erroneous 200 Responses	HTTP GET	20	20
	Erroneous 200 Responses	HTTP POST	20	20
	Valid 200 Responses	HTTP GET	20	20
	Valid 200 Responses	HTTP POST	20	20
	Identical 200 Responses	HTTP GET	8	8
	Identical 200 Responses	HTTP POST	8	8
	<i>False positive test cases</i>	HTTP GET	10	0
Local File Inclusion/Directory Traversal/Path Traversal	Erroneous 500 Responses	HTTP GET	59	59
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	59	59
<i>False positives reported = 0,00%</i>	Erroneous 404 Responses	HTTP GET	59	59
	Erroneous 404 Responses	HTTP POST	59	59
	Erroneous 200 Responses	HTTP GET	59	59
	Erroneous 200 Responses	HTTP POST	59	59
	Redirect 302 Responses	HTTP GET	59	59
	Redirect 302 Responses	HTTP POST	59	59
	Valid 200 Responses	HTTP GET	59	59
	Valid 200 Responses	HTTP POST	59	59
	Identical 200 Responses	HTTP GET	59	59
	Identical 200 Responses	HTTP POST	59	59
	<i>False positive test cases</i>	HTTP GET	8	0
Remote File Inclusion	Erroneous 500 Responses	HTTP GET	9	9
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	9	9
<i>False positives reported = 0,00%</i>	Erroneous 404 Responses	HTTP GET	9	9
	Erroneous 404 Responses	HTTP POST	9	9
	Erroneous 200 Responses	HTTP GET	9	9
	Erroneous 200 Responses	HTTP POST	9	9
	Redirect 302 Responses	HTTP GET	9	9
	Redirect 302 Responses	HTTP POST	9	9
	Valid 200 Responses	HTTP GET	9	9
	Valid 200 Responses	HTTP POST	9	9
	Identical 200 Responses	HTTP GET	9	9
	Identical 200 Responses	HTTP POST	9	9
	<i>False positive test cases</i>	HTTP GET	6	0

Unvalidated Redirect	Redirect 302 Responses	HTTP GET	15	15
<i>Coverage = 100,00%</i>	Redirect 302 Responses	HTTP POST	15	15
<i>False positives reported = 0,00%</i>	Valid 200 Responses	HTTP GET	15	15
	Valid 200 Responses	HTTP POST	15	15
	<i>False positive test cases</i>	HTTP GET	9	0
The Old, Backup and Unreferenced Files	Erroneous 404 Responses	HTTP GET	91	91
<i>Coverage = 100,00%</i>	Erroneous 200 Responses	HTTP GET	91	91
<i>False positives reported = 0,00%</i>	<i>False positive test cases</i>	HTTP GET	3	0
Coverage	99,52%			
False positive	0,00%			

A.2 Burp Suite

WAVSEP v1.5	Burp Suite v1.6.12			
Categories	Response Type	Method	Total	Detect
Reflected Cross Site Scripting (RXSS)	ReflectedXSS	HTTP GET	32	31
<i>Coverage = 91,18%</i>	ReflectedXSS	HTTP POST	32	31
<i>False positives reported = 0,00%</i>	DomXSS	HTTP GET	4	0
	<i>False positive test cases</i>	HTTP GET	7	0
SQL Injection	Erroneous 500 Responses	HTTP GET	20	20
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	20	20
<i>False positives reported = 30,00%</i>	Erroneous 200 Responses	HTTP GET	20	20
	Erroneous 200 Responses	HTTP POST	20	20
	Valid 200 Responses	HTTP GET	20	20
	Valid 200 Responses	HTTP POST	20	20
	Identical 200 Responses	HTTP GET	8	8
	Identical 200 Responses	HTTP POST	8	8
	<i>False positive test cases</i>	HTTP GET	10	3
Local File Inclusion/Directory Traversal/Path Traversal	Erroneous 500 Responses	HTTP GET	59	44
<i>Coverage = 69,49%</i>	Erroneous 500 Responses	HTTP POST	59	43
<i>False positives reported = 12,50%</i>	Erroneous 404 Responses	HTTP GET	59	29
	Erroneous 404 Responses	HTTP POST	59	28
	Erroneous 200 Responses	HTTP GET	59	44
	Erroneous 200 Responses	HTTP POST	59	43
	Redirect 302 Responses	HTTP GET	59	44
	Redirect 302 Responses	HTTP POST	59	43
	Valid 200 Responses	HTTP GET	59	44
	Valid 200 Responses	HTTP POST	59	43
	Identical 200 Responses	HTTP GET	59	44
	Identical 200 Responses	HTTP POST	59	43
	<i>False positive test cases</i>	HTTP GET	8	1
Remote File Inclusion	Erroneous 500 Responses	HTTP GET	9	7
<i>Coverage = 72,22%</i>	Erroneous 500 Responses	HTTP POST	9	7
<i>False positives reported = 0,00%</i>	Erroneous 404 Responses	HTTP GET	9	4
	Erroneous 404 Responses	HTTP POST	9	4
	Erroneous 200 Responses	HTTP GET	9	7
	Erroneous 200 Responses	HTTP POST	9	7
	Redirect 302 Responses	HTTP GET	9	7
	Redirect 302 Responses	HTTP POST	9	7
	Valid 200 Responses	HTTP GET	9	7
	Valid 200 Responses	HTTP POST	9	7
	Identical 200 Responses	HTTP GET	9	7
	Identical 200 Responses	HTTP POST	9	7
	<i>False positive test cases</i>	HTTP GET	6	0

Unvalidated Redirect	Redirect 302 Responses	HTTP GET	15	14
<i>Coverage = 46,67%</i>	Redirect 302 Responses	HTTP POST	15	14
<i>False positives reported = 22,22%</i>	Valid 200 Responses	HTTP GET	15	0
	Valid 200 Responses	HTTP POST	15	0
	<i>False positive test cases</i>	HTTP GET	9	2
The Old, Backup and Unreferenced Files	Erroneous 404 Responses	HTTP GET	91	23
<i>Coverage = 25,27%</i>	Erroneous 200 Responses	HTTP GET	91	23
<i>False positives reported = 33,33%</i>	<i>False positive test cases</i>	HTTP GET	3	1
Coverage	66,72%			
False positive	16,28%			

A.3. OWASP ZAP

A.3 OWASP ZAP

WAVSEP v1.5	OWASP ZAP v2.3.1			
Categories	Response Type	Method	Total	Detect
Reflected Cross Site Scripting (RXSS)	ReflectedXSS	HTTP GET	32	31
<i>Coverage = 92,65%</i>	ReflectedXSS	HTTP POST	32	32
<i>False positives reported = 0,00%</i>	DomXSS	HTTP GET	4	0
	<i>False positive test cases</i>	HTTP GET	7	0
SQL Injection	Erroneous 500 Responses	HTTP GET	20	20
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	20	20
<i>False positives reported = 0,00%</i>	Erroneous 200 Responses	HTTP GET	20	20
	Erroneous 200 Responses	HTTP POST	20	20
	Valid 200 Responses	HTTP GET	20	20
	Valid 200 Responses	HTTP POST	20	20
	Identical 200 Responses	HTTP GET	8	8
	Identical 200 Responses	HTTP POST	8	8
	<i>False positive test cases</i>	HTTP GET	10	0
Local File Inclusion/Directory Traversal/Path Traversal	Erroneous 500 Responses	HTTP GET	59	59
<i>Coverage = 77,12%</i>	Erroneous 500 Responses	HTTP POST	59	59
<i>False positives reported = 0,00%</i>	Erroneous 404 Responses	HTTP GET	59	59
	Erroneous 404 Responses	HTTP POST	59	59
	Erroneous 200 Responses	HTTP GET	59	59
	Erroneous 200 Responses	HTTP POST	59	59
	Redirect 302 Responses	HTTP GET	59	32
	Redirect 302 Responses	HTTP POST	59	32
	Valid 200 Responses	HTTP GET	59	32
	Valid 200 Responses	HTTP POST	59	32
	Identical 200 Responses	HTTP GET	59	32
	Identical 200 Responses	HTTP POST	59	32
	<i>False positive test cases</i>	HTTP GET	8	0
Remote File Inclusion	Erroneous 500 Responses	HTTP GET	9	9
<i>Coverage = 100,00%</i>	Erroneous 500 Responses	HTTP POST	9	9
<i>False positives reported = 16,67%</i>	Erroneous 404 Responses	HTTP GET	9	9
	Erroneous 404 Responses	HTTP POST	9	9
	Erroneous 200 Responses	HTTP GET	9	9
	Erroneous 200 Responses	HTTP POST	9	9
	Redirect 302 Responses	HTTP GET	9	9
	Redirect 302 Responses	HTTP POST	9	9
	Valid 200 Responses	HTTP GET	9	9
	Valid 200 Responses	HTTP POST	9	9
	Identical 200 Responses	HTTP GET	9	9
	Identical 200 Responses	HTTP POST	9	9
	<i>False positive test cases</i>	HTTP GET	6	1

Unvalidated Redirect	Redirect 302 Responses	HTTP GET	15	15
<i>Coverage = 100,00%</i>	Redirect 302 Responses	HTTP POST	15	15
<i>False positives reported = 22,22%</i>	Valid 200 Responses	HTTP GET	15	15
	Valid 200 Responses	HTTP POST	15	15
	<i>False positive test cases</i>	HTTP GET	9	2
The Old, Backup and Unreferenced Files	Erroneous 404 Responses	HTTP GET	91	35
<i>Coverage = 31,32%</i>	Erroneous 200 Responses	HTTP GET	91	22
<i>False positives reported = 100,00%</i>	<i>False positive test cases</i>	HTTP GET	3	3
Coverage	76,86%			
False positive	13,95%			

A.4 w3af

WAVSEP v1.5	w3af v1.6.48			
Categories	Response Type	Method	Total	Detect
Reflected Cross Site Scripting (RXSS)	ReflectedXSS	HTTP GET	32	25
Coverage = 79,41%	ReflectedXSS	HTTP POST	32	25
False positives reported = 0,00%	DomXSS	HTTP GET	4	4
	False positive test cases	HTTP GET	7	0
SQL Injection	Erroneous 500 Responses	HTTP GET	20	20
Coverage = 82,35%	Erroneous 500 Responses	HTTP POST	20	20
False positives reported = 30,00%	Erroneous 200 Responses	HTTP GET	20	20
	Erroneous 200 Responses	HTTP POST	20	20
	Valid 200 Responses	HTTP GET	20	14
	Valid 200 Responses	HTTP POST	20	14
	Identical 200 Responses	HTTP GET	8	2
	Identical 200 Responses	HTTP POST	8	2
	False positive test cases	HTTP GET	10	3
Local File Inclusion/Directory Traversal/Path Traversal	Erroneous 500 Responses	HTTP GET	59	33
Coverage = 39,97%	Erroneous 500 Responses	HTTP POST	59	32
False positives reported = 75,00%	Erroneous 404 Responses	HTTP GET	59	13
	Erroneous 404 Responses	HTTP POST	59	20
	Erroneous 200 Responses	HTTP GET	59	33
	Erroneous 200 Responses	HTTP POST	59	32
	Redirect 302 Responses	HTTP GET	59	20
	Redirect 302 Responses	HTTP POST	59	20
	Valid 200 Responses	HTTP GET	59	20
	Valid 200 Responses	HTTP POST	59	20
	Identical 200 Responses	HTTP GET	59	20
	Identical 200 Responses	HTTP POST	59	20
	False positive test cases	HTTP GET	8	6
Remote File Inclusion	Erroneous 500 Responses	HTTP GET	9	3
Coverage = 22,22%	Erroneous 500 Responses	HTTP POST	9	3
False positives reported = 33,33%	Erroneous 404 Responses	HTTP GET	9	0
	Erroneous 404 Responses	HTTP POST	9	0
	Erroneous 200 Responses	HTTP GET	9	9
	Erroneous 200 Responses	HTTP POST	9	9
	Redirect 302 Responses	HTTP GET	9	0
	Redirect 302 Responses	HTTP POST	9	0
	Valid 200 Responses	HTTP GET	9	0
	Valid 200 Responses	HTTP POST	9	0
	Identical 200 Responses	HTTP GET	9	0
	Identical 200 Responses	HTTP POST	9	0
	False positive test cases	HTTP GET	6	2

Unvalidated Redirect	Redirect 302 Responses	HTTP GET	15	11
<i>Coverage = 36,67%</i>	Redirect 302 Responses	HTTP POST	15	11
<i>False positives reported = 11,11%</i>	Valid 200 Responses	HTTP GET	15	0
	Valid 200 Responses	HTTP POST	15	0
	<i>False positive test cases</i>	HTTP GET	9	1
The Old, Backup and Unreferenced Files	Erroneous 404 Responses	HTTP GET	91	21
<i>Coverage = 23,08%</i>	Erroneous 200 Responses	HTTP GET	91	21
<i>False positives reported = 0,00%</i>	<i>False positive test cases</i>	HTTP GET	3	0
Coverage	42,55%			
False positive	27,91%			

Appendix B

Guide

Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

Security Testing in the CI Environment



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

Table of Contents

1	Introduction	3
2	Setup.....	4
2.1	endpoints.json.....	4
3	Report.....	5
4	Examples.....	7
4.1	endpoints.json.....	7
4.2	Jenkins.....	7



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

1 Introduction

This document is a small guide for Nordnet developers (employees and contractors) that wants to have their project tested for vulnerabilities in the CI environment.

Testing for vulnerabilities in the build process will hopefully lead to less vulnerabilities slipping out into production and hence protect our customers and Nordnet's reputation.

The actual setup is fast and will result in reports being created and available on Jenkins after the CI: Automatic tests phase is complete (see the Examples section). The purpose of the report is to give fast feedback on potential vulnerabilities to developers and to help QA with their manual work in the QA: Tests phase.

This document is divided into the following sections:

- Setup – This section explain how to modify your project to allow for testing in the CI: Automatic tests phase.
- Report – This section explain what is included in the report and why.
- Examples – This section contains examples of how it will look when everything is setup correctly, what kind of information you will find and how to interpret it.

For information on how to write secure code please refer to the Secure Coding Guideline and the References section of the report published to Jenkins.



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

2 Setup

To enable security testing on a project we use a vulnerability scanner. The scanner operates in two phases; Crawling and Penetration testing, where *crawling* means that the scanner tries to find all URLs of the application and *penetration testing* means that it tries to penetrate the application via these URLs, in order to discover security weaknesses.

The scanner needs to know where to start in order to crawl and test the application; therefore you **must** include a file with endpoints for your application, meaning URLs where the scanner can start.

2.1 endpoints.json

This is the file that you must include in your project. The file contains endpoint URLs (starting points for the scan) for the application you are setting up in the ci environment. The file should be in JSON format and look like this:

```
{
  "endpoints": [
    "http://target.ci/endpoint1",
    "http://target.ci/endpoint2"
  ]
}
```

The endpoint(s) must contain the full URL of the endpoint in the CI environment.

Note: The endpoints.json file should be located in:

```
<component_root>/test/security/ci/endpoints.json
```

For more information see the Examples section.



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

3 Report

When the vulnerability scanning is complete and vulnerabilities were found, a report is published on Jenkins in JUnit¹ format². In each such report the following items are included (see the Examples section for an example report):

- The category for the vulnerability as class name in the JUnit report. Since JUnit is built for Java³ and collects all issues per class this means that all issues in the same category are grouped together in the published report.
- URL - The URL of the vulnerability as both issue name in the JUnit report and as a field in the description. In the report all issues are named <classname>.<name>. Having the URL as name makes it easier to distinguish between issues.
- For each issue there is a description (mentioned in the above item about URL), this description includes the following:
 - Title - A title for the issue, a longer version of the category above, e.g. the category XSS_TAG has the name Cross-Site Scripting (XSS) in HTML tag - XSS in HTML tag.
 - Severity - The severity of the issue, this is any one of the following (sorted from lower to higher): informational, low, medium, high.
 - Description - A short description of the issue.
 - References - Some references to the issue from well-known sources such as OWASP⁴.
 - Variations - This is a list of all variations found by the scanner of how to exploit the specific vulnerability. The following is included for each variation:
 - Trust - If the variation is trusted or not (if it is untrusted it must be verified manually).
 - Affected page - A full URL of the exploit, works best when Method (below) is GET.
 - The description also include the following items, if they are available:
 - Injected - The injected payload.
 - Proof - Proof that the exploit worked.
 - HTTP request - The full HTTP request with all headers. This is a good complement to the affected page since this takes care of all HTTP methods (POST, PUT, etc.).

¹ <http://junit.org/>

² <https://svn.jenkins-ci.org/trunk/hudson/dtkit/dtkit-format/dtkit-junit-model/src/main/resources/com/thalesgroup/dtkit/junit/model/xsd/junit-7.xsd>

³ <https://www.java.com/en/>

⁴ https://www.owasp.org/index.php/Main_Page



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

- If applicable, the description also include the following items:
 - Method - The HTTP method used by the scanner when it found the issue (e.g. GET, POST etc.)
 - Parameter - The parameter that was vulnerable in this specific case.
 - Remediation Guidance - Some general information on how to remedy the vulnerability.



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

4 Examples

All examples in this section use the component *target_app*.

4.1 endpoints.json

Everything needed to setup your project is this file. Here is an example of how it looks in *target_app*.

Location:

```
target_app/test/security/ci/endpoints.json
```

Contents:

```
{  
  "endpoints": [  
    "http://target_app.ci/target_app/"  
  ]  
}
```

4.2 Jenkins

If you have finished the setup you will perhaps see something like this in Jenkins:

Build pipeline: target_app



Figure 1 - The build pipeline of *target_app*. This is what it looks like when the tests in the CI: Automatic tests phase has found issues and therefore marked the build as unstable.



Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

Clicking on an unstable (yellow/orange) box will bring you to the overview of that build phase:

Build baeec6f2f95.32
(2015-apr-24 09:52:35)

Started 5 days 4 hr ago
Took [1 min 25 sec](#) on master

Status

Changes
No changes.

Started by upstream project [target_app.CI](#) build number [35](#)
originally caused by:

- Started by upstream project [target_app.BUILD](#) build number [32](#)
originally caused by:
 - commit notification [baeec6f2f953efbb554f8379bb45b0d5abafb4e](#)

Revision: [baeec6f2f953efbb554f8379bb45b0d5abafb4e](#)

- detached

Test Result (3 failures / ±0)
[XSS.http://target_app.ci/target_app/test3/](#)
[XSS.http://target_app.ci/target_app/test2/](#)
[XSS_TAG.http://target_app.ci/target_app/test1/](#)

Figure 2 – The build information. Test Result is included both as a link and as an overview of each unstable build.

The build phase overview contains historical information about the testing as well (3 failures / ±0 means that the amount of failures is unchanged since the last build). Clicking on one of the links to Test Result will lead to the following page:

Test Result

3 failures (±0)



All Failed Tests

Test Name	Duration	Age
+ XSS.http://target_app.ci/target_app/test3/	24 sec	1
+ XSS.http://target_app.ci/target_app/test2/	14 sec	1
+ XSS_TAG.http://target_app.ci/target_app/test1/	54 sec	1

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
(root)	1 min 32 sec	3		0		4		7	

Figure 3 – Test Result overview. Contains information on how many test cases failed, passed or was skipped, among other things.




Security Testing in the CI Environment

Date	Author	File name	Version	Security classification
2015-06-22	Andreas Broström	Security Testing in the CI Environment	1.0	• Open

Test cases are named <Test category>.<URL> to make it easy to identify where an issue has occurred and what type it is. If one of the test cases is clicked it will lead to the following page, the major part of the report:

Failed

XSS:http://target_app.ci/target_app/test3/ (from security.severity.high)

Failing for the past 1 build (Since  #42)
[Took 24 sec.](#)

Error Message

XSS

Stacktrace

Cross-Site Scripting (XSS) - XSS

```
-----
* Severity: HIGH
* URL: http://target\_app.ci/target\_app/test3/
* Method: POST
* Parameter: username

#####
# Variations #
#####

Variation 1 (Trusted)
* Injected: ci_scanner_name<some_dangerous_input_374d8e30118449a9946323f960852279/>
* Proof: <some_dangerous_input_374d8e30118449a9946323f960852279/>

* Affected page: http://target\_app.ci/target\_app/test3/

* HTTP request:
POST /target_app/test3/ HTTP/1.1
Host: target_app.ci
Accept-Encoding: gzip, deflate
User-Agent: ci_scanner
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content-Length: 86
Content-Type: application/x-www-form-urlencoded

#####
# Description #
```

Figure 4 – The actual scan report. Contains the information described in section 3 – Report.



