

BACHELOR THESIS
COMPUTING SCIENCE



RADBOD UNIVERSITY

Security by design in Azure
DevOps pipelines, a case study at
SpendLab technology

Author:
Jesse van Son
s4601262

First supervisor/assessor:
prof. dr. ir. Joost Visser
j.visser@cs.ru.nl

Company supervisor:
Msc. Stijn Oostdam
stijn.oostdam@spendlab.nl

Second supervisor:
dr. ir. Erik Poll
erikpoll@cs.ru.nl



March 22, 2020

Abstract

This research investigates the impact of adding security tools in CI pipelines on “security by design” in software development. There are multiple ways of improving security by design in CI pipelines, and there is existing research in this area. However, not much research is done about actually measuring the improvements made.

This thesis starts off with a systematic review of existing approaches, where we conclude that static application security testing and open source security scanning are both methods used in CI pipelines to improve security by design. Secondly, we construct a Goal-Question-Metric model for measuring security improvements achieved by such pipeline extensions.

Finally, we conduct a case study at SpendLab Technology to test the model and evaluate how much security improved. For this case study software applications written in the .NET framework were measured. Azure DevOps was used for pipeline management. The software programs used for security scanning were WhiteSource Bolt and SonarQube. We collected measurement data generated by applying these tools to the software applications. We also interviewed the developers using them. From the case study, we learned that there are many practical obstacles that hinder the adoption of these security tools. While security improvements were detected, quantification of the improvements proved difficult, possible due to the limited scope and duration of the case study.

Contents

1	Introduction	4
1.1	Problem statement	4
1.2	Research goal	5
1.3	Research questions	5
1.4	Thesis outline	6
2	Preliminaries	7
2.1	Build pipelines	7
2.2	Azure DevOps	7
2.2.1	Branches	8
2.2.2	Commits and Pull Requests	8
2.2.3	Git Tag	8
2.3	NuGet packages and dependencies	8
3	Systematic review of existing approaches	9
3.1	Options for improving security in CI pipelines	9
3.1.1	Static application security testing	9
3.1.2	Open source security vulnerabilities check	11
3.1.3	Dynamic application security testing	12
3.1.4	Unit testing	12
3.2	Overview	13
4	Measuring improvement of security	14
4.1	Goal question metric template	14
4.2	Goal	15
4.3	Questions and corresponding metrics	15
4.3.1	Question: Are security problems detected with vulnerability scanning?	15
4.3.2	Question: Are security problems detected by the scanning useful?	17
4.3.3	Question: Do the security problems detected by the scans have effect?	18
4.4	Conclusion	20

5	Case Study at SpendLab Technology	21
5.1	Research Methodology	21
5.2	Evaluation of current pipelines in SpendLab Technology . . .	22
5.3	Setting up the case study	22
5.3.1	Additions to pipelines	23
5.3.2	Final pipeline definitions	27
5.3.3	Timeline of case study	27
5.3.4	First Meeting - 21 November 2019	28
5.3.5	Second Meeting - 11 December 2019	28
5.4	Measuring Security by Design Improvement	29
5.4.1	Metrics from GQM model	29
5.4.2	Questions from GQM model	35
5.4.3	Goal from GQM model	37
5.5	Impact on productivity	37
5.5.1	Time measurements from Azure	38
5.5.2	Interviewing developers	39
5.5.3	Conclusion	39
5.6	Completeness of measurements	40
6	Related Work	41
6.1	Existing research on security in pipelines	41
6.1.1	Evaluation and comparisons of tools	41
6.1.2	Surveys on SAST	42
6.1.3	Fault proneness SonarQube	42
6.1.4	Research on OSS	43
6.2	Delayed issue effect	43
6.3	Security of the pipeline itself	43
7	Conclusions	45
7.1	Methods for measuring security by design improvement . . .	45
7.2	Measuring security improvement at SpendLab	46
7.3	Recommendations for practitioners	47
8	Future Research	49
8.1	Performing more case studies	49
8.2	Dynamic Application Security Testing	49
8.3	Code quality improvement using SonarQube	49
8.4	Economics of security in CI pipelines	50
8.5	Optimizing SAST scanning	50
A	Appendix: OWASP versus WhiteSource	53
B	Appendix: Pipelines technical explanation	55
B.1	SpendLab Technology pipelines technical explanation	55

B.2	Final pipeline definitions	56
B.2.1	APRA Quickbuild.yml	57
B.2.2	APRA BuildForRelease.yml	57
B.2.3	Build-Framework-Solution.yml	57
C	Appendix: SonarQube setup	58
C.1	Configuration SonarQube	58
C.2	Disabled rules in SonarQube	59
D	Appendix: Interviews at SpendLab	60
D.1	SonarQube interviews	60
D.1.1	Questions	60
D.1.2	Developer 1	61
D.1.3	Developer 2	62
D.1.4	Developer 3	63
D.1.5	Developer 4	64
D.1.6	Developer 5	65
D.2	WhiteSource interview	67
D.3	Extra interview questions	68

Chapter 1

Introduction

1.1 Problem statement

SpendLab recovery has been conducting recovery audits for its public and private clients for decades. Such recovery audits discover incorrect cash-flows by manual analysis, involving large datasets with privacy sensitive data. SpendLab Technology, a subsidiary of SpendLab Recovery, is currently automating the recovery process. This involves a substantial software development effort. At SpendLab, Azure DevOps is used to manage the software team. The pipelines are defined using YAML files and are also managed from Azure DevOps.

Because of recent worldwide developments in privacy law and more general awareness of cyber security, it is of utmost importance for software development teams to develop secure software [10]. Developing secure software is in itself a very broad statement, and there are many things that need to be taken into account to make the claim that software is objectively secure.

A trend in developing secure software is security by design. This means that security should be taken into account while developing, and not in hindsight after the development process. If for instance a security flaw is discovered later on when the software has already been taken into production, this flaw will bring significantly more risk because attackers have had time to exploit it. A general rule of thumb has long been that issues that get solved later on in the development cycle are more costly to fix, but this is not always the case. More on this is discussed in section 6.2.

1.2 Research goal

Security by design can be implemented in many ways, and there are a multitude of options for companies to assure their company adheres to security by design principles. One of these options is extending the CI pipelines with security scanning. The pipelines are used to continuously build and integrate new parts of the software. An example of an extension in the pipeline is scanning for security risks from outdated dependencies.

The goal of this research is to investigate how we can measure increases in security by design when improving CI pipelines. There already has been substantial research about trying to make software more secure by improving CI pipelines [7][11][19]. There is, to our knowledge, not much research on to what extent additions in pipelines have effect. This thesis will be about a research problem in design science. It aims to answer a knowledge question about security by design in CI pipelines in the context of software companies.

1.3 Research questions

At SpendLab technology, continuous integration tools like build pipelines are used. The goal as stated in section 1.2 leads to the following research question:

RQ: To what extent can security by design be improved through inserting security checks into CI pipelines?

To answer this question, some way of measuring security improvements must be developed. For this we have a sub-question:

RQ.a: How can security by design improvement through inserting security checks in CI pipelines be measured?

We will also study whether such security checks are actually improving the security using our measurement method developed in RQ.a.

RQ.b: Can increases in security by design through improving CI pipelines actually be measured in practice?

RQ.b will be answered through a case study at SpendLab technology.

1.4 Thesis outline

In chapter 2, the preliminaries are discussed, which are needed to understand this research. Then, in chapter 3 there will be a systematic review of existing approaches, which will provide a lot of background information and foundation for the research in measuring improvements. In chapter 4, we provide research on how to measure improvement. Then, chapter 5 discusses a case study conducted at SpendLab technology using the findings of chapter 4. Chapter 6 contains related work. Finally, chapter 7 finishes with a conclusion and chapter 8 contains potential future research.

Chapter 2

Preliminaries

This chapter describes some terms used throughout this thesis that may not be known to every reader. If you know about the terms this chapter can be skipped.

2.1 Build pipelines

As mentioned in the introduction, Azure DevOps CI/CD (Continuous integration/Continuous Delivery) pipelines are used to manage building software. Continuous integration means that new code is frequently integrated with the existing code, for instance through Pull Requests which are explained later in this chapter. During this integration the code is compiled to make sure nothing is broken, and sometimes running automated (integration) tests is also part of CI. Continuous Delivery means that there is always a tested and working product ready to deploy. There are CD pipelines that build and deploy the application to test servers automatically. Pipelines are very powerful tools that can be extended to include much more functionality, which is researched in this paper in the context of security.

2.2 Azure DevOps

Azure DevOps is software made by Microsoft to support software teams. It is very similar to GitHub and is used by SpendLab Technology to manage the team and the software they create. The functionalities used by SpendLab include Git repositories, Pipelines, Agile Boards and Test Plans.

2.2.1 Branches

Within a Git project there is often one main branch of code where everybody works from, this could be called *develop*. Developers can create new branches based on this *develop* branch, and write new code there. When the new code is done, they can merge their branch with the *develop* branch. This is mainly done so multiple programmers can work simultaneously on one application.

2.2.2 Commits and Pull Requests

Commits and Pull Requests are ways of uploading code to the Git server at Azure DevOps. A commit is done to a branch, and you upload part of the code to the server when you push the commit. Pull Requests are made to merge two branches. If a developer wants to merge his branch with the main *develop* branch, he creates a pull request. This pull request can be reviewed by colleagues to check the work according to the four eye principle. If all code is deemed correct the pull request can be accepted.

On every commit or pull request it is possible to run a pipeline, this pipeline could for instance build and test the code.

2.2.3 Git Tag

Commits in Git can be tagged by a developer, this is mostly used to mark the tagged point as important. Git Tags are used at SpendLab to mark a release. A pipeline can be triggered on a specific commit by tagging that commit.

2.3 NuGet packages and dependencies

In essence, NuGet is a platform for developers to create, share and consume code and is created by Microsoft for .NET. In the context of this thesis, NuGet is a package manager that manages dependencies for SpendLab Technology. If a package is installed for a project, NuGet automatically installs other packages that are needed for the first package. All these packages are dependencies of a project.

Chapter 3

Systematic review of existing approaches

In this chapter, a systematic review of possibilities for inserting security checks into CI pipelines will be conducted. The possibilities mentioned in this chapter all have some relevance to security by design.

3.1 Options for improving security in CI pipelines

There are multiple ways to improve Security by Design, but in this thesis the focus will be on security by design improvement in CI pipelines. In figure 3.1 an overview is given of the options for security in the development process, of which the CI pipeline is a part, at SpendLab Technology. At the end of this chapter, a choice will be made between which security validation methods will be included in this research.

First, static application security testing (SAST) will be discussed. Then we will look at open source security vulnerabilities (OSS). We will also discuss the options for dynamic application security testing (DAST) and unit testing. Integration testing is omitted because the relevance to security is deemed low, and it is mostly used in the CD part of pipelines.

3.1.1 Static application security testing

Static Application Security Testing is one way of testing for application security in CI pipelines. A SAST tool works on the principle of white box testing, the tool has all the code and can analyse the given code to see if any paths bring security risks with it. SAST scanners can help with discovering

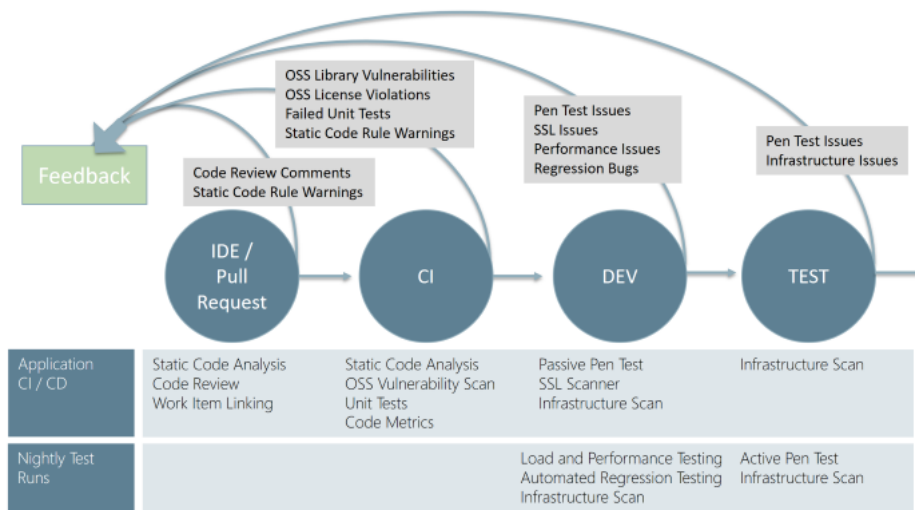


Figure 3.1: Options for adding security validations in the development process by Microsoft [1]

bugs or vulnerabilities early, so they can be resolved before releasing. These scanners can be implemented in the pipelines, but can also be executed within the IDE by a developer.

Disadvantages

One of the main disadvantages of scanner tools like these are false positives [11]. It is difficult to find a balance between showing all possible errors, and not showing false positives. Naturally, a tool wants to give all possible security flaws by showing as many potential errors as possible, but this comes with a risk of showing more false positives. The danger in false positives with a SAST tool might be that eventually developers take the tool less serious. Weeding through false positives takes time and is a tedious task.

Options

OWASP has a compiled list with all options for SAST Solutions [5]. For the case study, Azure DevOps and C# will be used, so our main options are limited to the following well known tools:

- Puma Scan
- SonarQube

- Checkmarx
- Veracode

For the case study, SonarQube was chosen as the SAST scanner.

3.1.2 Open source security vulnerabilities check

Open Source Software (OSS) has long been seen as very secure, because everybody can check the source code. However, blind trust in Open Source Software is never a good idea. In a study conducted by Guido Schryen [21] it was concluded that: "open source and closed source software development do not significantly differ in terms of vulnerability disclosure and vendors' patching behavior, a phenomenon that has been widely assumed, but hardly investigated." This makes clear that open source software is not necessarily more secure than closed source software.

It is clear that open source software cannot be trusted blindly, this is why a vulnerability check for open source software is an option for the development cycle. These OSS security scanners work by looking at all the packages used by an application, for instance NuGet packages in .NET, and compare the versions to their databases. These databases contain known vulnerabilities for dependencies. If there are known vulnerabilities, this is brought up by the scanner.

Disadvantages

The biggest disadvantage with these scanners is again false positives. For the same reasons as named in section 3.1.1, false positives are a big deciding factor for determining the quality of an OSS tool. These false positives could be that vulnerable packages are found, but the actual vulnerable parts are never used in the code.

Another decision that has to be made when using this software is when to scan for packages. One possibility is to scan the final built solution, where a scanner will only find the libraries (.dll files) that will be used. Another possibility is scanning the whole source directory of a project. If the whole directory is scanned, the scanner will not be sure if all found dependencies are actually used. This is because frameworks like .NET bring a lot of dependencies and packages, which might in the end not be used by the compiled code. If we scan the compiled solution, we will only find packages that are actually used, but we will also miss a lot of other potential unsafe dependencies. This is because the front end packages, for instance bootstrap, are not added through .dll files. These front end JavaScript packages are

just added as files to the solution, and thus are not found when scanning the compiled solution.

Options

SpendLab technology uses NuGet to manage packages. From the options found there are two not too expensive possibilities that integrate with Azure DevOps and support NuGet. These options are WhiteSource Bolt [6] and OWASP dependency check[4]. In the case study we decided to go with WhiteSource Bolt, for which a reasoning can be found in Appendix A. In short, OWASP returned many false positives and found less true positives than WhiteSource Bolt.

3.1.3 Dynamic application security testing

Dynamic application security testing (DAST) is a form of black box testing. White box testing, for instance SAST scanners, have the source code available to scan. DAST tools finds vulnerabilities by actually performing attacks on the compiled code. DAST tools can make an addition to SAST tools, to have a more complete automated security testing package.

Disadvantages

A DAST scan does not necessarily find all vulnerabilities, it just tries a predefined list of attacks on the known attack surface by the scanner. From recent research conducted in 2018, where tools were tested against a test project from OWASP, we can see that not even close to all issues were found by DAST scanners [16]. They do help with finding some issues and make a start to developing a more secure application, as can be seen in the cited research.

Options

There are a multitude of options listed by OWASP [2]. Almost all of them are commercial, but we found the most documentation and research on the OWASP Zed Attack Proxy (ZAP) project.

3.1.4 Unit testing

Unit testing can also be important for security. Unit tests make sure that for every code path, the correct results are given. This could be relevant to

security, if for instance a user must have certain privileges to access data. If an application does not have unit tests, it might be harder to fix security issues because you do not know if you are breaking any functionality when applying the fix.

Most companies have some policy that unit tests have to be written for specific parts of code, and the correctness and completeness is mostly checked in the review process. The completeness of unit tests could however often be overlooked, and that is where unit test coverage scanners come into play. These scanners can analyse if all code paths are tested in some way. It can, however, not tell if the method of testing shows that it is secure. Because of this, and to limit the scope of this thesis, we decided not to take unit testing into account.

3.2 Overview

Multiple options for security by design have been discussed. Table 3.1 gives an overview of the possibilities and at which stage of the pipeline they can be applied. The approval stage is the stage where the lead developer gives the green light for the application to go to the test servers. The test servers are used by the testers to test new functionality and report back with issues before the application goes to production.

	CI		CD
	Build pipeline	Approval stage	Test servers
Unit Tests	x	x	
SAST	x	x	
OSS vulnerability	x	x	
DAST			x

Table 3.1: Overview of security options in CI/CD pipeline

The build pipeline and approval stage are on the side of continuous integration, and the test servers are on the side of continuous delivery. Because the scope of this thesis is only concerned with CI and security related tests, we will only look at SAST and OSS vulnerabilities. Unit Tests are omitted because only automated coverage testing is not deemed security relevant enough.

Now that the ways of adding security to CI pipelines have been discussed, the next chapter will concern itself with finding a way to measure security improvement in our CI pipelines.

Chapter 4

Measuring improvement of security

To be able to quantify security improvement, we first need to research a method of measuring security improvements. There are multiple ways to structure this, and we have chosen to create a Goal Question Metric (GQM) model to show how improvement can be measured [8]. A GQM model starts with a template to specify the goal, then questions are formulated that can be used to answer the goal. Finally, metrics are coupled to these questions.

4.1 Goal question metric template

For the GQM model, we will first create a template like given in [8]. The template for our GQM model is given in table 4.1.

Purpose	Improve
Issue	the security by design
Object	through security scanning in the CI pipeline
Viewpoint	From the technical manager's viewpoint

Table 4.1: GQM template

The purpose from the viewpoint of the technical manager within SpendLab Technology is improving the security by design through security scanning in the CI pipeline. The objective of this research is to measure if we are actually improving security by design.

4.2 Goal

Using table 4.1, a goal can be formulated, from which the questions and metrics will origin. The goal of this model will be: **Improvement of security validation in CI pipelines.** This will be looked at from the technical manager's viewpoint. At SpendLab, the technical manager is responsible for the security of the delivered product. He must be able to defend that security by design has been considered and has been improved by the implementation of different tools in the CI pipelines.

4.3 Questions and corresponding metrics

To answer the main goal, several sub-questions are needed. These sub-questions can be coupled to one of the extensions added to the CI pipeline described in chapter 3. The questions are all described in subsections of this chapter. To answer the questions, metrics are needed. The metrics needed to answer one of the questions are described together with the question. A graphical overview of the GQM model is given in in figure 4.1.

A symmetry between the first and last column can be seen, where the first question row concerns vulnerable libraries, the second question row concerns outdated libraries, and the final row concerns internal vulnerabilities.

4.3.1 Question: Are security problems detected with vulnerability scanning?

The first question might be obvious, but is necessary to say something about security improvement. If no problems are caught by the scanner, we need to evaluate if the application is perfect, or if there might be false negatives.

Metric: Number of vulnerable libraries discovered by OSS scanner

To say something about caught problems by scanning, we can look at the number of vulnerable libraries found by the OSS scanner. A higher number of vulnerable libraries would logically indicate that the scanned application is less secure.

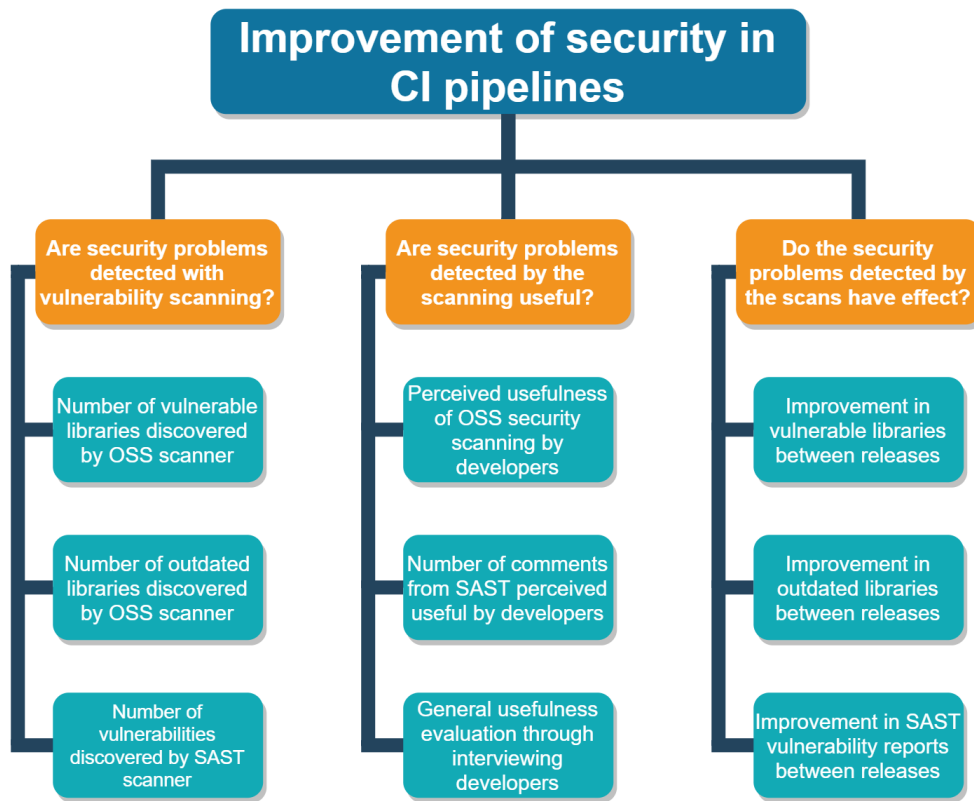


Figure 4.1: Goal Question Metric diagram

Metric: Number of outdated libraries discovered by OSS scanner

Outdated libraries can also be a security risk. A study by Cox et al. shows that systems with many outdated libraries are more than four times as likely to contain security issues [12]. This is because outdated libraries are more likely to have vulnerabilities for which exploits have been developed.

Metric: Number of vulnerabilities discovered by SAST scanner

For the SAST scanner to be perceived as useful, it must first actually find vulnerabilities. If there are no vulnerabilities found, it might be that the scanner does not do a proper job. It could also be that the application is completely secure, which is highly unlikely.

4.3.2 Question: Are security problems detected by the scanning useful?

The previous question in itself is not enough to answer the main goal. To say something about actual improvement, we first have to determine if the detected problems are useful. There are many metrics from the tools we use which might be able to help answer this question, these are described below.

Metric: Perceived usefulness of OSS scanning by developers

With this metric we want to measure how useful developers found OSS scanning after using it for some amount of time. The metric will be answered through a survey, using a Likert scale [18]. The statement is: "OSS scanning is useful within our company", and the scale would have the options:

1. Strongly disagree
2. Disagree
3. Neither agree nor disagree
4. Agree
5. Strongly agree

Metric: Number of comments from SAST perceived useful by developers

With this metric we want to measure how useful developers found SAST scanning after using it for some amount of time. A SAST can place comments on a pull request, which thus easily integrates with the CI process. The usefulness could be answered by analyzing every comment from a SAST, and having a developer evaluate the comment to useful or not useful. Another option is through a survey, using a Likert scale. This would be done in the same fashion as above, but with the statement: "The comments made by the SAST are all useful"

Metric: General usefulness evaluation through interviewing developers

The two earlier metrics were part of a bigger interview, where some open questions were asked to do qualitative research. This was done to obtain

more useful data, because there are only 6 developers working at the company used for the case study at the time of writing.

There are three options for conducting these interviews: structured interviews, semi-structured interviews and unstructured interviews. For the case study in chapter 5, semi structured interviews will be conducted. this means some questions will be prepared to spark discussion and get relevant feedback [18]. The asked questions are defined in Appendix D. The interview starts with some questions about the experience and role of the interviewee, and after that some more in depth questions about SonarQube or WhiteSource Bolt.

4.3.3 Question: Do the security problems detected by the scans have effect?

To answer the final goal, it is important to know if the detection of problems leads to actual improvements. The first question answers if problems are found and how many are found. From the second question, we know how useful the found issues are. Finally, we need to look at the improvements in the reports from OSS and SAST scans, and determine if the measures taken in the CI pipelines have effect. The significance of these effects can be answered using the previous question.

Metric: Improvement in vulnerable libraries between releases

Most OSS security scanners generate some form of a report. For OSS security scanning WhiteSource Bolt is used in the case study. An example can be found in figure 4.2.

If an issue is found, there is a severity level coupled to this issue. This metric shows the number of issues found at every release, and shows trends in number of issues over time. If issues are not solved, we must look into why this is the case. If issues are resolved but new ones keep popping up, there could be some research as to why there are new issues.

Metric: Improvement in outdated libraries between releases

As can be seen in figure 4.2, WhiteSource shows how many dependencies are outdated. Outdated dependencies could be a security risk, because newer libraries often feature security updates. It could be that security flaws in outdated libraries are not yet found by OSS security scanners, these false negatives are a big risk to security, because they could already be known by attackers.



Figure 4.2: Example of part of a report generated by WhiteSource Bolt

With this metric, a trend of improvement or stagnation can be shown. If WhiteSource Bolt is implemented, it could be that there would be improvement in the number of outdated dependencies. It could also be that this does not happen, and more dependencies become outdated and nobody feels obliged to update these libraries.

Metric: Improvement in SAST vulnerability reports between releases

Most SAST tools generate some kind of report from a scan. This report shows certain metrics, from which we can make conclusions on security improvement in general. For our Case Study at SpendLab technology we will be using SonarQube as a SAST. An example of a SonarQube report generated is shown in figure 4.3.

There are 4 sections in the dashboard: bugs and vulnerabilities, code smells, coverage, and duplications. For security by design, we will mostly be looking at the bugs and vulnerabilities section. This metric will be answered by creating a zero measurement, described in the case study. From there scans

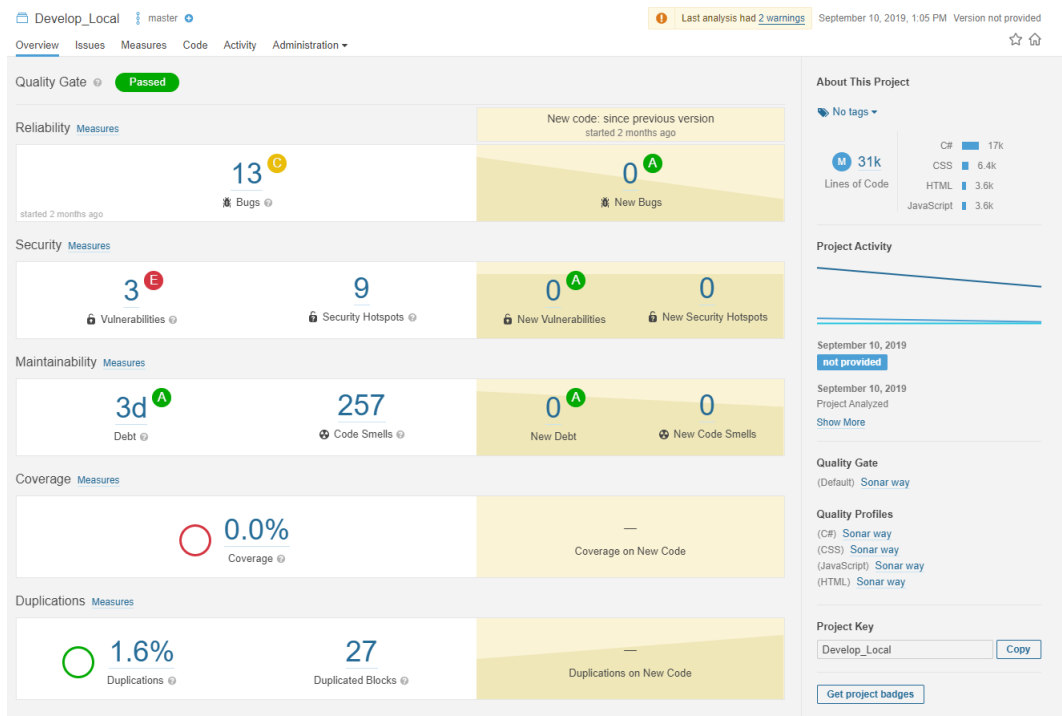


Figure 4.3: Main dashboard of a project analyzed by SonarQube

will be done at every release, and we can try to extract a trend from these figures.

4.4 Conclusion

We now formulated a goal, which we will answer through our identified questions and metrics. This GQM metric model was created to answer RQ.a: How can security by design improvement by inserting security checks in CI pipelines be measured? In the next chapter this GQM model will be used to answer if security is improved at SpendLab Technology.

Chapter 5

Case Study at SpendLab Technology

The previous two chapters described the options for taking security measures in the CI pipeline, and how to measure security improvement. Now this builds up to this case study, where we will try to measure security improvement at SpendLab Technology.

SpendLab Technology is a software team working for SpendLab Recovery, and creates workflow software to analyze financial data. At SpendLab Technology, Azure DevOps is used to manage the software team. The pipelines are defined using YAML files and are also managed from Azure DevOps. The main application created by SpendLab Technology is Advance Payable Recovery Audit (APRA) together with the Database Quality Gate (DBQG), APRA is the front end and the DBQG is used for back end processing. This case study will focus on APRA and the DBQG, but the developed methods can later be used on other software projects at SpendLab Technology or any other company.

5.1 Research Methodology

An embedded case study will be conducted, which means that the case study contains more than one sub unit of analysis in the same context. The context is SpendLab Technology, the case is the improvements in the CI pipelines, and the sub units of analysis will be APRA and the DBQG [20]. The main research question for this case study will be: can increases in security by design through improving CI pipelines actually be measured in practice?

5.2 Evaluation of current pipelines in SpendLab Technology

Before the changes made with this thesis, with every commit or pull request, APRA and DBQG run a QuickBuild to check if the application builds on the server and if all unit tests are successful. A graphical overview can be found in figure 5.1.

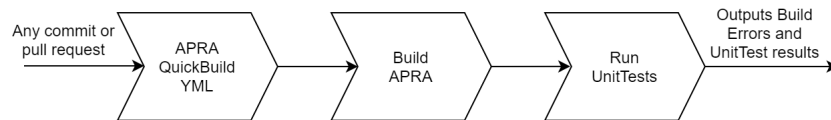


Figure 5.1: QuickBuild of APRA or DBQG

If a commit or pull request is tagged with a git tag called "workflow" for APRA or "dbqg" for the DBQG, which is mostly done on the main develop branch, the Build for Release pipeline will run. After running the pipeline the release process is started. This is visually displayed in figure 5.2. After the last acceptance phase, the new version of the software is sent to production.

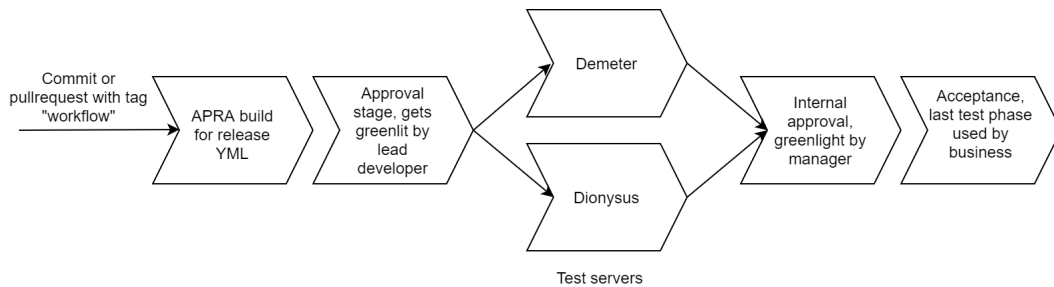


Figure 5.2: Build for release of APRA and DBQG

The exact way the QuickBuild.yml and BuildForRelease.yml work at SpendLab is described in Appendix B.

5.3 Setting up the case study

Now the current state of affairs at SpendLab is explained, the set up of the case study is layed out here. First, it is shown how security checks have been added to the pipeline. Then a timeline is given of the case study, and last there is a report on how two meetings went where the additions to the pipelines were discussed.

5.3.1 Additions to pipelines

From chapter 3, it is concluded that we will be adding a SAST scanner and OSS scanner. For the SAST tool SonarQube Developer edition was selected, this was chosen because people within SpendLab Technology already had experience with this tool. The tool is also inexpensive and easy to use and integrate with Azure DevOps. For our OSS security scanning we chose WhiteSource Bolt. A reasoning for why we chose WhiteSource can be found in appendix A.

About SonarQube

SonarQube is a well known software quality scanner. It has multiple uses, and combines all in a dashboard with an intuitive overview. We set up SonarQube on a Azure Virtual Machine, using a Microsoft SQL studio database.

How SonarQube works

There are multiple ways to integrate SonarQube into the development process. We could manually scan the application from time to time. With the developer edition we can scan with every pull request, or we can scan with every build for release. In figure 5.3, a visualization is shown of where the scans will be done. The SonarQube scans will be ran for pull requests and release builds. This is done by using a trigger in the QuickBuild, which only runs the SonarQube related steps on a pull request. More technical explanations can be found in Appendix B.

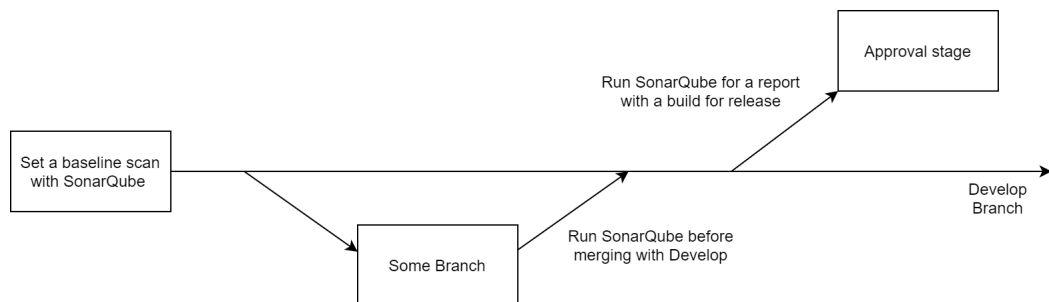


Figure 5.3: Figure showing where SonarQube is triggered in the CI process

Exclusions in analysis

With SonarQube, it is possible to exclude certain parts of a solution from analysis. At SpendLab the folders with JavaScript libraries are excluded, because these are not maintained by SpendLab and it saves lines of code to scan. The migrations folder is also excluded, because this is automatically generated code. These exclusions also save some time by not scanning the code.

Useful rule set

Rules are used by SonarQube to determine if some piece of code is an issue. SonarQube has a standard rule set called the *sonar way*. This contains all rules SonarQube deems useful, but some rules might not be found necessary by the developers at SpendLab Technology. If this is the case, we will dynamically create a new rule set tweaked for use at SpendLab Technology. This new rule set is called the *SpendLab way*.

Rule types and severity

SonarQube distinguishes three types of issues: bugs, vulnerabilities and code smells. For every rule it is categorized in one of these issues, and a severity tag is given to the rule. The severity tag has the following options:

1. Info
2. Minor
3. Major
4. Critical
5. Blocker

The category and the severity of a rule are customizable for every rule by the user. This was not done for this research, because it was not needed and would not have influenced the results significantly.

SonarQube caveats

While using SonarQube at SpendLab, some caveats were found that hamper the measurements for this case study. In section 4.3.2, we described a metric used to measure how serious developers take SonarQube comments. The first plan was analyzing all comments placed by SonarQube on pull requests, and see if developers resolve these comments or if they resolved them as won't

fix. If developers did not want to fix a problem, we would ask them to explain why in a comment. This information would be used to disable rules that SpendLab found unnecessary. Ideally, a percentage of comments taken serious by developers could be made from this. Because of shortcomings of SonarQube this was not possible. Not all problems found by the scanner were actually commented, which might be a result of the API not allowing that many comments by an external source.

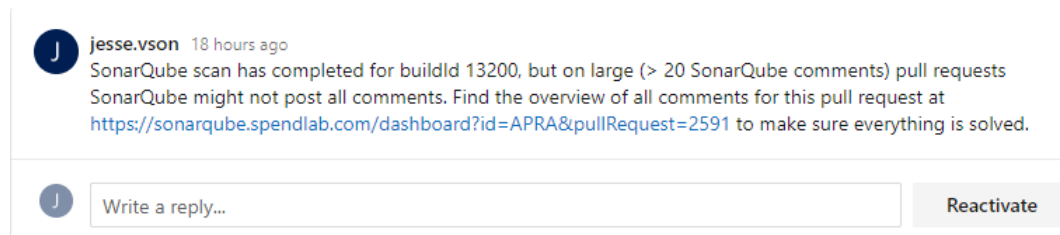


Figure 5.4: Figure showing the workaround for SonarQube not placing all comments

This issue was solved by automating a comment with the REST API of Azure DevOps, the comment shown in figure 5.4 is placed on every pull request. This way developers can easily find all issues, which can be used as a workaround. The comment is placed as a resolved comment, so the comment does not interfere with completion of a pull request. When looking at the GQM model, this does influence the question: "Are security problems detected with vulnerability scanning?". Problems are found, but not always properly reported.

Another problem was that when a problem is solved in a pull request before it is merged to the main branch, SonarQube deletes this issue and never counts it as one of the issues on the main branch. This made it very hard to backtrack the problems found by SonarQube. An example can be found in figure 5.5. This is a branch analysis of a pull request related to user management with 193 file changes. As can be seen, 4 vulnerabilities were found. The hard coded password vulnerabilities were deemed false positives, and the tainted data that was logged is fixed with a sanitize function.

Because SonarQube does not log issues fixed in the pull request, it is hard to measure all solved issues for this research. This caveat influences RQ.b: "Can increases in security by design through improving CI pipelines actually be measured in practice?". Because of this issue, measuring improvements with SonarQube becomes difficult.



Figure 5.5: Figure showing 4 vulnerabilities found by SonarQube

Setting up WhiteSource Bolt

Our OSS check will only be ran after a build for release. This is because in the scanned software systems there are almost no pull requests containing new libraries, and the new libraries added are mostly not outdated. To set up we installed WhiteSource Bolt [6], a convenient extension for Azure DevOps that can be added to the YAML of a pipeline. It will automatically run if a pipeline is triggered and then it will generate a report that can be found in the Azure DevOps environment.

When WhiteSource Bolt scans the directory containing all source files, it finds 678 dependencies to scan and 5 vulnerable ones. With only the folder containing the compiled application we find 240 dependencies, and only 1 vulnerable package. A reasoning for this difference can be found in section 3.1.2.

There has to be made a choice between these two options, and for now we will use the whole sources directory and not only the publish folder. This is done because otherwise we might not find critical security flaws in front end JavaScript packages.

WhiteSource Bolt caveats

WhiteSource Bolt has some issues that influence this research. As mentioned above, a lot of doubles are found by WhiteSource Bolt. One analysis from the 10th of december on APRA found 680 vulnerable libraries. If we remove all exact duplicates using the library name, we only have 401 libraries left. After doing this there are still some more duplicates, for instance one .dll file found and the same package as .nupkg file. WhiteSource support was mailed, and they explained that NuGet dll's are a unique case where the

same library can be resolved from different binary files which have the same name and different hash values. This is because they are compiled under different framework versions, and they are thus treated as different libraries by WhiteSource. Because of the presence of duplicate findings, we can not really put value in exact numbers measured by WhiteSource Bolt. We can however look at the changes relative to earlier measurements.

Another caveat is that if WhiteSource should find false positives, we are not able to suppress them using WhiteSource Bolt. Because of this and the doubles, the WhiteSource Bolt interface is cluttered. This influences the GQM model question "Are security problems detected by the scanning useful?", because issues reported multiple times are not useful.

Lastly, WhiteSource does not find all front end packages. This is due to SpendLab not using a package manager like NPM for front-end JavaScript packages. Lodash and Handlebars are both examples of packages that were vulnerable in APRA, but not found by WhiteSource Bolt. This influences the GQM model question: "Are security problems detected with vulnerability scanning?", because issues that could be there are not always found by WhiteSource Bolt.

5.3.2 Final pipeline definitions

In section 5.2, we examined how the pipelines work at SpendLab. To implement SonarQube and WhiteSource Bolt, we made some changes to these pipelines. Performance was also taken into account, because performance influences usability. The main change is that a SonarQube prepare and scan task are added to both the QuickBuilds and BuildForReleases. The QuickBuild used to be executed on every update of a branch and a pull request, but running on every update was deemed unnecessary and this was changed to only run on pull requests. The BuildForRelease also includes a WhiteSource Bolt scan on the Sources folder. More technical changes and the YAML files can be found in Appendix B.2.

5.3.3 Timeline of case study

First of all, baseline scans with SonarQube and WhiteSource Bolt were made to have a reference point. The plan executed for measuring improvement was to first start with a meeting. In this meeting a short presentation would be held about SonarQube and WhiteSource, and how it is implemented within our company. From there we would start using the software and gather data for this research. The timeline of the research is visualized in figure 5.6.

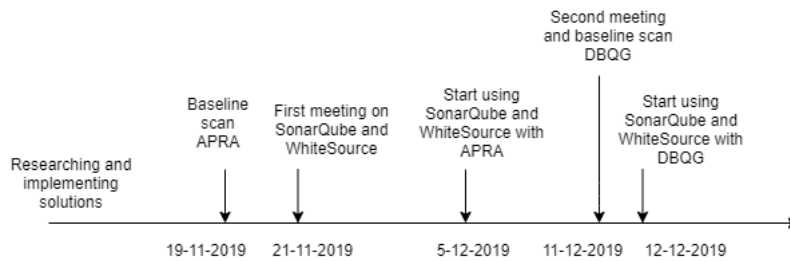


Figure 5.6: Timeline of events described in this case study

Within the company it was chosen to implement SonarQube and WhiteSource for the DBQG later, to first make sure everything works correctly on APRA. This is why the baseline scan for the DBQG is not at the same time as APRA.

5.3.4 First Meeting - 21 November 2019

The first meeting was held on 21 November 2019, all six developers working on the software were present, except one who was briefed later. The meeting went well, and there was one important point raised for this research. WhiteSource seems to only find packages installed through NuGet, and not all front end JavaScript dependencies. An example of this is the JavaScript package Bootstrap. This is very important to take into account for determining the scope and correctness of this research.

During the meeting, it became clear that developers were agreeing with the issues found by the scanners. They saw it as a duty of notifying management of these security risks, and seemed to be motivated to fix these problems. To look into the effectiveness of just notifying developers, and not forcing action, we decided to wait until the second meeting and see if anything would happen.

5.3.5 Second Meeting - 11 December 2019

From the above meeting on, people were aware of the security flaws in APRA. The comments placed in pull requests by SonarQube got resolved, but not the backlog of problems already there. To continue on the previous meeting, we wanted to discuss in a quick meeting how we would continue with the current known issues. Here it became clear that the security problems have priority, and would be placed on the backlog. Because SpendLab was working towards a big demo version of the application, fixing the SonarQube and WhiteSource problems got a lower priority. After the release of this demo version, fixing security issues would get higher priority.

The decision to wait has been made because most of the security risks could be mitigated, while the application is still only used within the company and is not accessible for everyone.

5.4 Measuring Security by Design Improvement

Now that the set up of the case study is discussed, this chapter will look into measuring the security by design improvement using the GQM model constructed in chapter 4. First all metrics will be measured, then the questions will be answered using these metrics and finally we will try to reach the goal of this measurement process.

5.4.1 Metrics from GQM model

To answer the three questions from our GQM model, all metrics defined in chapter 4 will be measured here. Using figures we created from data gathered at SpendLab, we will answer multiple questions.

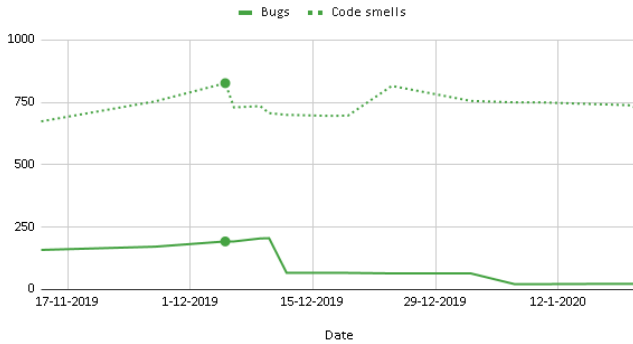
Data from SonarQube

First, we can create a graph with number of issues over time. This graph, for APRA, can be found in figure 5.7. There are some notable points in this graph. The first date is the baseline scan. The point at the 5th of December, which is marked in the graph, is the point where SonarQube is implemented in the pipelines, so from here we would ideally expect that not a lot of issues are introduced into the software.

After the 5th of December, many bugs seem to disappear. This is not always because people were fixing issues found by SonarQube, but because we disabled some rules used to scan the solution. The disabled rules can be found in Appendix C.2. This does not threaten the validity of this research, because the disabled rules were only for code smells which are irrelevant to security. There are not many issues added after the implementation of SonarQube, except many code smells the 24th of December. This was because a large pull request was merged, but not all code smells were resolved. The reason for this was that not all comments were placed by SonarQube as described in section 5.3.1.

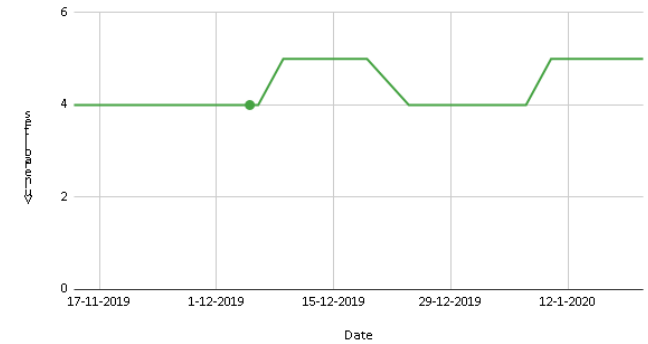
The data for the DBQG can be found in figure 5.8, the data here is less interesting because this is a smaller project with less development in the time span of this case study. There is also no data from before the implementation of SonarQube in the pipelines. From the 27th to the 29th, many issues have

APRA bugs and code smells over time



(a) Bugs and code smells

APRA vulnerabilities over time

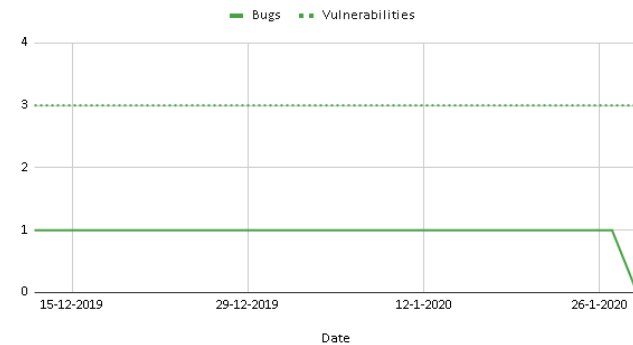


(b) vulnerabilities

Figure 5.7: Shows SonarQube statistics for APRA

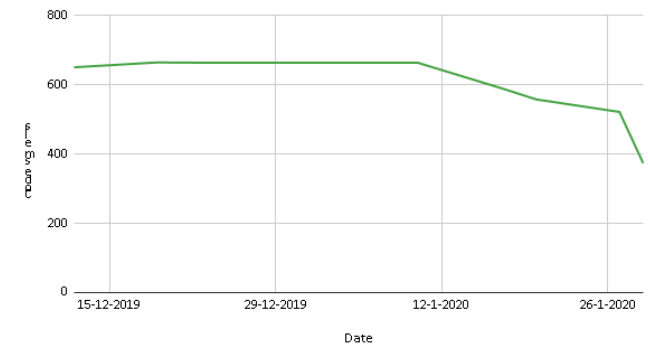
disappeared. This is because big parts of unused code were deleted and other parts of the application were refactored in one pull request. This was done with help of SonarQube, thus there were no new issues introduced because they were resolved in the pull request.

DBQG bugs and vulnerabilities over time



(a) Bugs and vulnerabilities

DBQG code smells over time



(b) code smells

Figure 5.8: Shows SonarQube statistics for DBQG

In the figures we also included bugs and code smells. While these are less relevant to security than vulnerabilities, they are still useful to study how serious developers find the tool. If developers do not take bugs or code smells serious, they are unlikely to take vulnerabilities serious.

Using these figures we can answer the following metrics:

1. Number of vulnerabilities discovered by SAST scanner:
SonarQube reports 5 vulnerabilities for APRA and 3 for DBQG.

2. Improvement in SAST vulnerability reports between releases:
There seems to not be much improvement outside of the disabled rules. There is however a noticeable stagnation, not much new vulnerabilities or bugs are added to the solutions. As mentioned in section 5.3.1, vulnerabilities solved in the pull requests are not counted in figure 5.7 and 5.8. Together with our observation from 5.3.1 on SonarQube caveats, we can conclude that there is improvement in the handling of security issues found in applications at SpendLab.

Perceived usefulness of comments placed by SAST

Because of the problems described in section 5.3.1 about SonarQube caveats, this metric is answered through a Likert scale as described in section 4.3.2. The interviews conducted and some more in-depth analysis can be found in Appendix D. In figure 5.9 we see the results of our Likert scale.

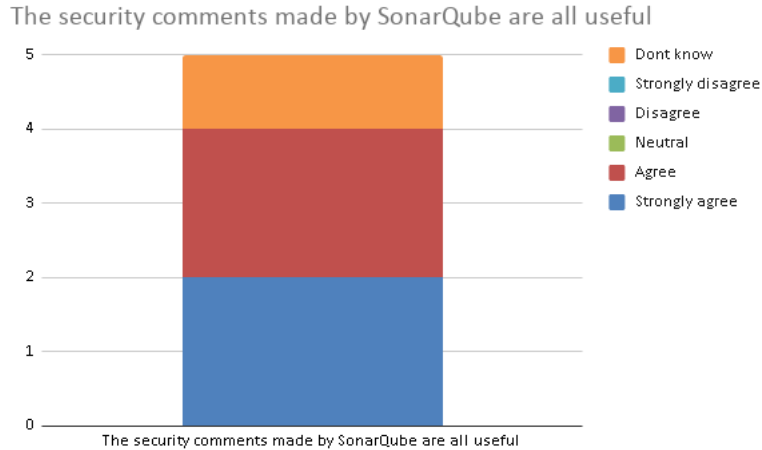
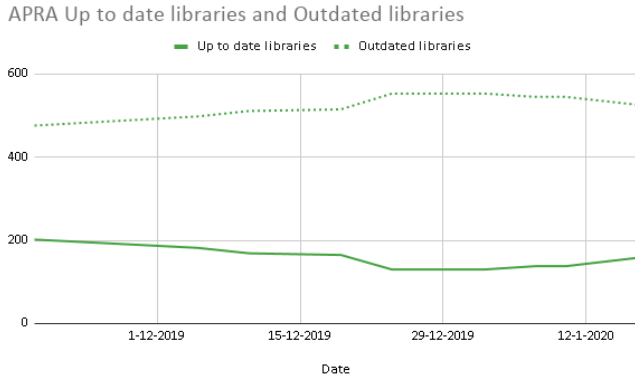


Figure 5.9: Figure showing results of survey question about usefulness of SonarQube security comments

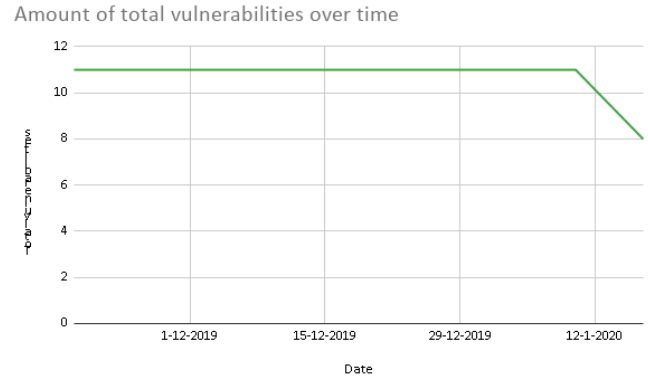
From this survey, a conclusion can be made that developers at SpendLab think that SonarQube comments are a useful addition to their workflow. The developer that filled in *don't know* explained that he did not know enough about the SonarQube rule set and did not get enough comments about security to answer this question properly.

Data from WhiteSource Bolt

For WhiteSource Bolt, graphs with results found over time can be found in figure 5.10.

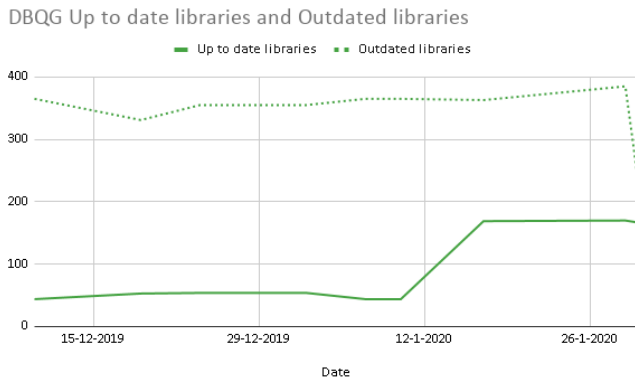


(a) Op to date and outdated libraries

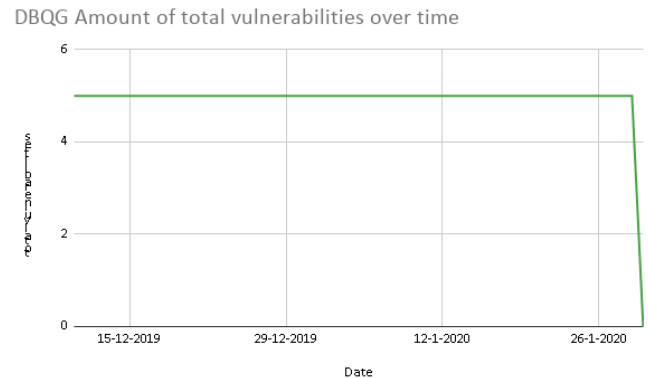


(b) Total number of vulnerabilities found in libraries

Figure 5.10: Shows WhiteSource Bolt statistics for APRA



(a) Op to date and outdated libraries



(b) Total number of vulnerabilities found in libraries

Figure 5.11: Shows WhiteSource Bolt statistics for DBQG

As mentioned in section 5.3.1 about WhiteSource caveats, the absolute numbers in these graphs are inaccurate due to duplicates. The general trend noticed in the graph is that the outdated libraries are growing in number, which demonstrates that nobody is actively monitoring these libraries. From the 7th of January, it seems that the up to date libraries are growing in numbers. But this is only by a small amount, and the majority of dependencies are still outdated.

Another observation is the improvement in the report of 17 January. This

is because one developer looked at the WhiteSource report and did an one-time bulk repair effort. This resulted in the following change log in a pull request for APRA and the DBQG:

- Update Microsoft.Data.OData to fix vulnerability CVE-██████████
- Update jQuery to ██████ to fix vulnerability CVE-██████████
- Update handlebars to ██████ to fix vulnerability CVE-██████████
- Update Lodash to fix vulnerability CVE-██████████
- Update Bootstrap and PopperJS
- Update jQuery validate

As explained in 5.3.1, Lodash and Handlebars were both vulnerable but not found by WhiteSource Bolt. The developer fixing these issues knew about them from his own private projects analyzed by GitHub's built-in security alerts for open source packages, where he did use a front-end package manager.

The developer looking into these issues noted that some major issues found were not directly in our own project, something obscure was is going on within NuGet that we cannot directly solve. Also the vulnerability in OData still pops up, while this library was updated.

When looking into the logs, it was found that WhiteSource Bolt still found the older OData package on the build server. After verifying that this package was definitely gone locally, extra research was done on why WhiteSource found the older packages. We found out that when a build is done on the server, the SourcesDirectory is not automatically cleaned. More information on how the build server exactly works can be found in Appendix B. After setting up the pipelines to clean up the SourcesDirectory, 145 fewer packages were found for the DBQG. The DBQG publish folder went from 406 to 251 mb. This does explain the dip in updated and outdated libraries shown in graph 5.11 on the 30th of January. The same was tried for APRA, but this did not yield significant results.

Using these graphs we can answer the following metrics:

1. Number of vulnerable libraries discovered by OSS scanner:
As was argued above, the number of vulnerable libraries is not exact because of duplicates. But there are vulnerable libraries found, which is the most important observation with this metric.
2. Number of outdated libraries discovered by OSS scanner:
There are outdated libraries discovered by WhiteSource Bolt, but the exact number is incorrect for the same reason as above.

3. Improvement in vulnerable libraries between releases:
Over the time the vulnerable libraries were measured at SpendLab, there was an improvement in reducing the vulnerable libraries.
4. Improvement in outdated libraries between releases:
From figure 5.10 we see that the outdated libraries are increasing over time. This is not an improvement for security, and a policy on updating libraries might be a good addition to SpendLab Technology.

Perceived usefulness of OSS scanning by developers

With WhiteSource Bolt, we were planning on using the same Likert scale as SonarQube to answer the perceived usefulness by developers. This is in the end not possible due to the lack of data, because only one developer looked into WhiteSource Bolt.

From the interview with this developer, which can be found in Appendix D.2, we can conclude that OSS scanning is found useful by him. He does note that there are usability issues with WhiteSource Bolt as a tool, but the general usefulness of OSS security scanning is clear.

General usefulness evaluation through interviewing developers

To look into the general usefulness of the additions to the CI pipeline, interviews with the developers using the CI pipelines were held. Here, some conclusions will be given that could be drawn from the interviews. As mentioned earlier, full summaries of the separate interviews can be found in Appendix D.

The first thing that occurred to most developers when asked about the comments SonarQube placed, was that they agree with the comments. However, they do not always agree with the severity or the labeling as a vulnerability. This was because the issues found by SonarQube were very context dependent, and not always a security risk. The developers did note that it is still good the comments were placed, because it forced them to look at it. The developers all said that they take the comments seriously, and investigate them when they are placed under their pull request.

Developers also found that sometimes, the description of a rule in SonarQube did not directly say why something would be considered a vulnerability.

Again, a reaction to a statement was asked to plot the opinions on SonarQube. The statement was: SonarQube improves the security of applications

developed at SpendLab. The plotted results can be found in figure 5.12 below.

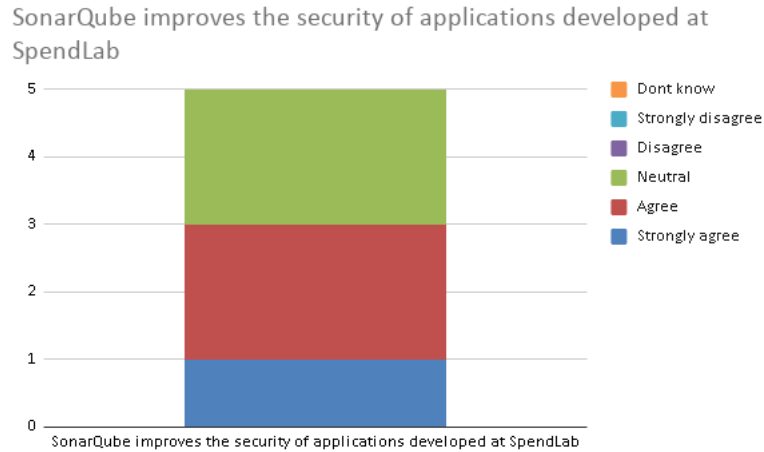


Figure 5.12: Results of survey question about general security improvement of applications developed at SpendLab.

As we can see in this figure, some developers are neutral to this statement. They explained that they did see the added value from SonarQube, especially in code quality and bugs. The improvement in security was according to these developers not really clear, because they did not get many security issues through SonarQube. And if they got security issues, they were sometimes not sure why it was a security issue and not just a bug or code smell. The developers agreeing mostly did this on the basis that it was good that people thought more about security, so as an awareness tool, and that security by design was more involved in the development process.

During the interviews, many developers noted the impact of SonarQube on build times. This was seen as an impact on productivity, and was not originally thought of in the GQM model. We decided to add section 5.5 to shed some further light to this, in addition to the rest of this case study.

5.4.2 Questions from GQM model

Now to answer the main goal, all metrics answered above will be used to answer the three questions asked in the GQM model from chapter 4.

Are security problems caught with vulnerability scanning?

The answer to this question is simply yes, we did find security issues through the vulnerability scanning additions to the CI pipelines. With the next two questions we will evaluate these found problems.

Are the problems caught by the scanning useful?

This question will be answered by looking at the results of the metrics under it.

- Perceived usefulness of OSS security scanning by developers:
The developer using OSS security did find it useful, but had some remarks on usability and false positives.
- Number of comments from SAST perceived useful by developers:
All developers agreed that the comments from a SAST that were security related were useful.
- General usefulness evaluation through interviewing developers:
While developers did see the comments as useful, they did not see many security related comments on their code. They also could sometimes not see why something was a security issue and not a code smell.

Taking all the metrics into account, we can say that in general, the consensus is that security problems detected are useful. There are however, many footnotes together with this claim, which can be found in the sections corresponding to the metrics above.

Do the problems caught by the scans have effect?

Again, the metrics belonging to this question are itemized below with their final conclusion.

- Improvement in vulnerable libraries between releases:
It is concluded that there is some improvement in vulnerable libraries, because one developer created a pull request with updates for libraries.
- Improvement in outdated libraries between releases:
The outdated libraries seem to steadily rise, and are not kept up to date by SpendLab. A update policy is not in place, and might be a good idea for SpendLab.
- Improvement in SAST vulnerability reports between releases:
The SAST vulnerability report did not see much improvement, because developers have not yet gone back to fix older issues. There are no

new issues introduced because they get taken seriously and solved in the pull requests. Because of this deviation from the trend, we can definitely say that security is improving through SAST.

Looking at the metrics, we can conclude that there is not much improvement, but in general there is a stagnation and the problems are not getting worse. Because we answered with the previous question that the found vulnerabilities are in general useful, we can conclude that the answer is yes, the problems caught by the scans do have effect. There are again remarks to be made with this conclusion, which can be found in the sections corresponding to the metrics above.

5.4.3 Goal from GQM model

The main goal set out by the GQM model was improvement of security in CI pipelines. We hoped to be able to improve security through the addition of SAST scanning and OSS security scanning in the CI pipelines at SpendLab.

Looking at all the questions, we can conclude that the goal is achieved, the security is improved through taking measures in the CI pipelines. There are however many remarks made, and a list of possible improvements and advises should be made to help future implementations improve. These remarks are named throughout the metrics part of this case study, and are summed up in the conclusion in section 7.3.

In the introduction, we noted that RQ.b would be answered through this case study. The research question was as follows: Can increases in security by design through improving CI pipelines actually be measured in practice? We would conclude that the improvement in security is measurable through our GQM model. The consensus seems to be that the security of applications developed at SpendLab did improve, but we did not fully succeed to quantify how much we improved. More on this is explained in the conclusions in section 7.2.

5.5 Impact on productivity

As has been discussed in section 5.4.1 about the general usefulness, impact on productivity was a big point for many users. Because this was not exactly in the scope of the thesis, but is very interesting nonetheless, there is some extra short research in this section.

5.5.1 Time measurements from Azure

Because of our additions to the CI pipelines, the time it takes for builds to complete is longer, and this can cause irritation. Azure has functionality where it shows statistics from pipelines, here we can look into the time measurements of QuickBuild and BuildForRelease for APRA. We decided to not take the DBQG into account because it yields similar results and there is more data on APRA.

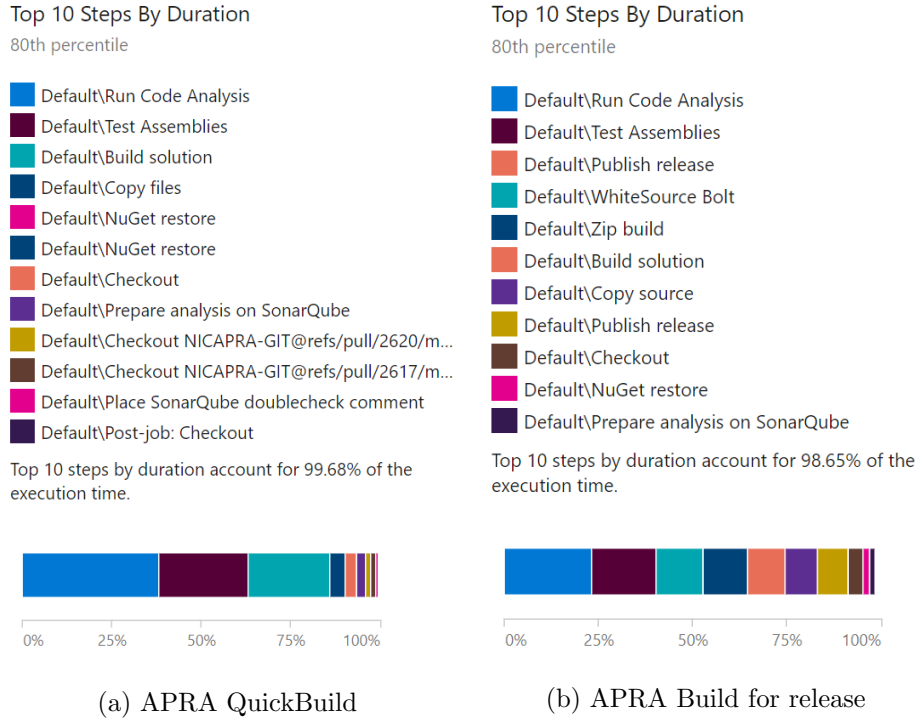


Figure 5.13: Shows the top 10 steps by duration for APRA from 28-12-2019 to 27-01-2020 (30 days)

On average, the Build for Release takes 24 minutes and 6 seconds, and the QuickBuild takes 13 minutes and 34 seconds. The statistics in figure 5.13 show that the *run code analysis*, the main task executed for SonarQube, takes 23.21% (around 5 minutes and 36 seconds) of the time for the Build for release and 38.32% (around 5 minutes and 12 seconds) for the QuickBuild. WhiteSource Bolt takes 12.41% (around 3 minutes) of the time for a build for release.

5.5.2 Interviewing developers

In the second interview conducted, which can be found in Appendix D.3, we asked developers if the extra time needed for building the pipelines is justified by the gain SonarQube gives in security. The results can be found in figure 5.14.

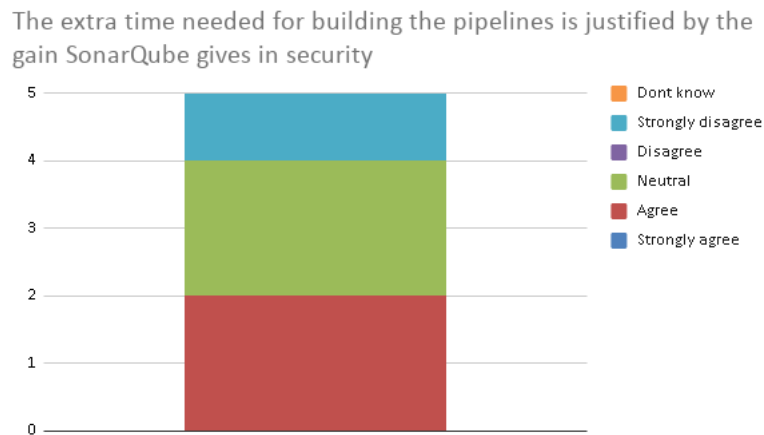


Figure 5.14: Results of survey question about the extra time used for builds by SonarQube being justified or not.

We can see there is divide in opinions on this statement. The developer strongly disagreeing with the statement argued that the security rules from SonarQube were useful according to him, but because he did not get any security risks in his code it is not worth the trade off in time when seeing SonarQube from a security point of view. The agreeing or neutral developers thought that it is worth it, but there is room for improvement in implementation or optimization of the usage of these tools.

5.5.3 Conclusion

As has been shown in this section, there is a significant impact on execution times of pipelines by the security additions to the CI pipelines. The developers have mixed feelings about this and are not entirely sure if the extra build time is always worth it.

An argument in favor of the CI pipelines could be made that developers already had to wait, the few extra minutes are therefore not another obstruction of the workflow. The step just takes some time longer, and it does increase the security of developed applications in the long run.

A further possibility would be optimizing the scanning, by for instance omitting certain low risk folders to save scanning time. Lastly, we could look into improving the performance by changing when SonarQube is ran.

5.6 Completeness of measurements

There are a few remarks to be made about the completeness of the measurements. While SpendLab Technology was a good example case for this thesis, it is a very small team. Because only 5 developers were interviewed, the closed questions have maybe too little input to extract a meaningful result. Because WhiteSource Bolt was only used briefly by one developer, there is even less input on the OSS scanning side. This might influence the generalizability of this research.

Chapter 6

Related Work

As mentioned in the introduction, the work related to this thesis will be discussed in this chapter. We will start with looking into existing research on this topic to our knowledge. Then more research is discussed that has relevance and is interesting together with this thesis.

6.1 Existing research on security in pipelines

In this chapter we will look into existing research on security in pipelines, and what is already known in addition to this research.

6.1.1 Evaluation and comparisons of tools

Most research done on SAST are evaluations of currently existing tools, or a comparison of tools by using an existing vulnerable application as test bed [7]. This cited research made a comparison between some SAST tools and concludes that SAST helps with uncovering some security issues, but it is not enough to uncover all weaknesses.

There is also a case study at Telenor Digital, where it is described how they integrated SAST, benchmarked it, and evaluated it in the company [19]. This study concluded that there are barriers in the tools performance and developers perceptions, like low performance in finding security issues and non-functional nature of security bugs. They did find that teams were still positive to use SAST tools to reduce security bugs, which is in line with this research.

In this thesis there was no thorough research on which tool to use, because we took the technology choices of the host company as a given. Nonetheless,

this is still an interesting and relevant research topic. To make a decision which tool in the end will be the best choice, many factors must be taken into account. Some interesting ones are cost, integration possibilities, ease of use and performance of the tool.

One example of research or a tool for this topic is the OWASP benchmark project, a free and open test suite designed to evaluate the speed, coverage and accuracy of automated software vulnerability detection tools and services [3].

The DAST tools discussed in the systematic review can also be tested using the OWASP benchmark. One example is research by Mburano and Si, where they test Arachni and OWASP zap, both SAST scanners, against the WAVSEP benchmark and the OWASP benchmark [16]. They found that both tools excelled in different categories, and there is not a single tool that does it all as of writing that thesis.

6.1.2 Surveys on SAST

Christakis and Bird performed an empirical study at Microsoft, where they found through interviews and surveys what developers want and need from program analysis [11]. The most relevant to this research that they found that security issues were ranked number one on what developers would like detected from SAST scanners. When asked about known security related incidents and if developers thought they could have been caught by SAST scanners, only 47% of the developers said yes. Which means that less than half of the developers trust a SAST tool to find security bugs that they found themselves.

6.1.3 Fault proneness SonarQube

Looking at SonarQube, which is used in our case study, there is also research done on the fault proneness of bug rules [15]. This study by Lenarduzzi et al concludes that many rules that are classified as bugs by SonarQube have a low fault-proneness, which means that the bugs found by SonarQube will not quickly introduce new faults in the software. They also say that the severity of a bug is not related to fault-proneness, and therefore severity should not be taken too serious as decision factors by developers looking into refactoring a violation.

6.1.4 Research on OSS

On the performance of OSS scanners itself there is not much research to our knowledge. However, there is research on impact of security risks in open source libraries [13]. This cited research by Decan et al concludes that 15% of the vulnerabilities found in OSS are fixed after their publication or not fixed at all. This means that developers were informed of the issue and when it would be published, but they did not fix it in time. OSS scanners also look into outdated libraries. According to research, software that uses many out of date libraries is four times more likely to contain security issues in these dependencies [12]. This validates that it is important to look into OSS security.

6.2 Delayed issue effect

An assumption in software development has long been that delayed issues get harder to fix over time. This is called the delayed issue effect. This assumption is used to justify the use of CI pipeline scanning tools, because issues are found quicker and are thus easier to solve.

According to a study by Tim Menzies et al, almost all research that claims this effect dates back to Barry Boehm and his book Software Engineering Economics from 1981 [9]. The study by Menzies et al found that in general they could not observe the delayed issue effect in any of the cases analyzed by them [17]. This is mostly credited to the fact that present day software development is often an agile process, where changes are more easily made. They also say that the delayed issue effect may continue to be prevalent in some cases, such as high-assurance software, architecturally complex systems, or in projects with poor engineering discipline.

In this thesis, CI pipeline scanning tools are used for security reasons. It is important that issues are found and solved, we do not specifically look into the easiness of solving issues by finding them earlier.

6.3 Security of the pipeline itself

This research mainly concerns itself with security in pipelines, but what of the security of the pipeline itself? If the software added into the pipeline is not secure, it could potentially be a security risk.

There is a master thesis by Michael Koopman that tries to create a framework to create a baseline which the company can use to detect and prevent

security vulnerabilities in their platform [14]. He concludes that there are risks associated with CI/CD pipelines. Controls were identified to mitigate each threat, and there were risk levels assigned to all threats.

Chapter 7

Conclusions

In this chapter all conclusions that can be drawn from the preceding chapters will be presented. The research question of this thesis was: to what extent can security by design be improved through inserting security checks into CI pipelines?

To answer the question above this thesis started with a systematic review, where an attempt has been made at finding all ways of adding security checks in pipelines. In the end it was concluded that we would limit the scope to implementing Static Application Security Testing (SAST) scanners and Open Source Software (OSS) security scanning.

7.1 Methods for measuring security by design improvement

Next, chapter 4 concerned itself with finding a way to measure security by design improvement. This chapter was used to answer RQ.a: How can security by design improvement through inserting security checks in CI pipelines be measured? We decided on creating a goal question metric (GQM) model to measure security by design improvement, where the goal formulated was improvement of security in CI pipelines. This goal would be answered through three questions and nine corresponding metrics. The metrics try to define security through amount of vulnerabilities found and solved through pipeline tools, and through interviewing developers about security at SpendLab since installing the pipeline extensions.

7.2 Measuring security improvement at SpendLab

Finally, RQ.b was answered. The question was: can increases in security by design by improving CI pipelines actually be measured in practice? A case study at SpendLab Technology, a subsidiary of SpendLab Recovery, was conducted to put the GQM model to the test. In the end, it was concluded that the goal of improving security in CI pipelines was achieved. To what extent the improvements helped security by design is debatable, because this is a tricky business with many variables. This thesis did not fully succeed to define exactly how much security improved.

There are many reasons encountered during this thesis that explain why quantification of the effect of tools is difficult. Below, these reasons are summarized.

- Difficult to install and configure tools.
- Unexpected behaviour from tools (see section 5.3.1):
 - Problems are not properly reported on pull requests by SonarQube.
 - SonarQube reports on main branch, not what happens inside pull requests.
 - Non suppressible false positives in WhiteSource Bolt.
 - Double vulnerabilities and packages found by WhiteSource Bolt.
 - Not all dependencies are found by WhiteSource Bolt because no front-end package manager is used.
 - Old remaining dependencies on the build server are not cleaned up by default.
- Severity indicators are inaccurate and not very useful.
- Long running time of the tools.
- False positives and false negatives.
- Not clear how issues of different types (internal vulnerability, vulnerable dependency, outdated dependency) should be aggregated.

From the GQM model we could see that security was defined by amount of vulnerabilities found by scanners, and this would be supplemented by interviews with developers. To say something about the amount of vulnerabilities found, you must know how much vulnerabilities there are in total. If this is known for an application, a percentage of vulnerabilities found by tools can be deduced.

There were not many vulnerabilities found in total. Because statistics tend to get more accurate with more data, it was hard to get an accurate result from this case study. This could be fixed by having larger projects or increasing the amount of subjects for this case study. It could also be that the project baseline quality at SpendLab Technology was very high, which means that not many vulnerabilities were found.

7.3 Recommendations for practitioners

During this research, many pitfalls and footnotes were discovered when implementing the security additions in the CI pipelines. In addition to the conclusions given above, a list with recommendations is given here. This summarizes many of the smaller problems, caveats and tips found throughout the case study.

- Do not underestimate the time it takes to set up tools correctly. This was a pitfall discovered with this thesis, one example being the cleaning up of the sources directory for WhiteSource described in 5.4.1.
- Use tools with a clear interface and good usability. WhiteSource Bolt had some problems with doubles, false positives and it could not suppress issues.
- Make sure an update policy for the libraries is in place. WhiteSource Bolt is a good tool for showing issues in dependencies, but is not of great help with fixing these issues. The interviewed developer noted that according to him, it would be a good idea to have someone look into updates for dependencies at least once a month.
- Try to tweak the rule-set of a SAST as well as possible. If developers see many issues they do not agree with, the tool will be taken less serious.
- Make sure the tools integrate well with the developer environment. As was seen with SonarQube, at SpendLab we ran into the issue that not all found vulnerabilities were automatically commented on pull requests. This was resolved as described in section 5.3.1.
- Try to minimize impact on runtime of pipelines. This could be done by not scanning when it is not necessary, and not scanning certain parts of the application that have a near zero probability of finding significant errors.
- Do not take the severity of rules coupled by some SAST tools too serious or as a heavy deciding factor. As mentioned in section 6.1.3, and in our interviews, the severity is mostly not correct.

In the research by Christakis and Bird [11] that was also used in section 6.1.2, there is a list of pain points reported by developers when using program analysers. If we cross reference this with our recommendations, many of these pain points were also encountered. Wrong checks are on by default, false positives, too slow, difficult to fit into workflow, selectively turn off analysis, and ranking of warnings are all addressed in this recommendations list.

Chapter 8

Future Research

8.1 Performing more case studies

This research only conducted one case study, and thus had limited data available. It could be that case studies at different companies yield different results, maybe due to other work culture or a more experienced team. To draw a more meaningful conclusion, multiple case studies should be conducted using the GQM model and recommendations from this thesis.

8.2 Dynamic Application Security Testing

For the scope of this thesis, DAST scanners were not taken into account. These scanners are however very important to supplement the static scanners. Further research could be to include those scanners in the GQM model, and to see if they are beneficial for security. This would change the research from CI pipeline additions to CI/CD pipeline additions.

8.3 Code quality improvement using SonarQube

With this thesis, only security related comments were taken into account. As is mentioned throughout the case study, SonarQube also places comments about bugs and code smells. Developers did note in their interviews that they see these comments as very helpful. It was concluded that because of this, SonarQube is taken more seriously, and thus developers would take security comments more seriously. Further research could be done to look into how much code quality improves by using SAST scanners like SonarQube or if developers like these tools.

8.4 Economics of security in CI pipelines

In section 5.5, a small start has been made on the economics of security. There was some discussion if the extra time it costs to fix the issues found by SonarQube were worth it. To make this discussion more complete, there could be research into the economics of security. An useful addition would for instance be not only interviewing the developers, but also interviewing the stakeholders or product owners.

8.5 Optimizing SAST scanning

As has been noted in section 5.5, the SAST scans take some time. Research could be done on how to improve the performance of scanning. This could be through framework changes, or measuring how much difference a faster server makes. There could also be research in optimizing settings with for instance not scanning certain low risk parts of an application to improve performance.

Bibliography

- [1] Add Continuous Security Validation to your CICD Pipeline | Microsoft Docs. <https://docs.microsoft.com/en-us/azure/devops/migrate/security-validation-cicd-pipeline?view=azure-devops> (visited on 07-11-2019).
- [2] Category:Vulnerability Scanning Tools - OWASP. https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools (visited on 22-10-2019).
- [3] OWASP Benchmark. <https://owasp.org/www-project-benchmark/> (visited on 29-01-2020).
- [4] OWASP Dependency Check - OWASP. https://www.owasp.org/index.php/OWASP_Dependency_Check (visited on 07-11-2019).
- [5] Source Code Analysis Tools - OWASP. https://www.owasp.org/index.php/Source_Code_Analysis_Tools (visited on 22-10-2019).
- [6] WhiteSource Bolt - Visual Studio Marketplace. <https://marketplace.visualstudio.com/items?itemName=whitesource.ws-bolt> (visited on 07-11-2019).
- [7] H. H. AlBreiki and Q. H. Mahmoud. Evaluation of static analysis tools for software security. In *2014 10th International Conference on Innovations in Information Technology (IIT)*, pages 93–98, November 2014.
- [8] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The Goal Question Metric Approach. 1994.
- [9] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, N.J, 1 edition edition, November 1981.
- [10] Andrew Burt. New Laws on Data Privacy and Security Are Coming. Is Your Company Ready? *Harvard Business Review*, July 2019.
- [11] Maria Christakis and Christian Bird. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the*

- 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM. event-place: Singapore, Singapore.
- [12] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring Dependency Freshness in Software Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118, May 2015. ISSN: 1558-1225.
- [13] A. Decan, T. Mens, and E. Constantinou. On the Impact of Security Vulnerabilities in the npm Package Dependency Network. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 181–191, May 2018.
- [14] Michael Koopman. A framework for detecting and preventing security vulnerabilities in continuous integration/continuous delivery pipelines, June 2019.
- [15] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. Are SonarQube Rules Inducing Bugs? *arXiv:1907.00376 [cs]*, December 2019. arXiv: 1907.00376.
- [16] B. Mburano and W. Si. Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark. In *2018 26th International Conference on Systems Engineering (ICSEng)*, pages 1–6, December 2018.
- [17] Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. Are Delayed Issues Harder to Resolve? Revisiting Cost-to-Fix of Defects throughout the Lifecycle. *Empirical Software Engineering*, 22(4):1903–1935, August 2017. arXiv: 1609.04886.
- [18] Briony J. Oates. *Researching Information Systems and Computing*. SAGE, 2006.
- [19] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 86–103, Cham, 2018. Springer International Publishing.
- [20] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, December 2008.
- [21] Guido Schryen. Is Open Source Security a Myth? *Commun. ACM*, 54(5):130–140, May 2011.

Appendix A

Appendix: OWASP versus WhiteSource

To make a small comparison between WhiteSource Bolt and OWASP Dependency Scan for the case study conducted in 5, we executed them both on the software used in the case study. With OWASP we found 6 vulnerable dependencies:

1. Microsoft.Owin.Security.Facebook.dll
This .dll is used by Microsoft as middleware that enables an application to support Facebook's OAuth 2.0 authentication workflow. OWASP tried to recognise it as Facebook PhotoUploader 4.5.57.0, which is outdated and insecure. This is thus also a false positive.
2. Microsoft.AspNet.SignalR.SystemWeb.dll
3. Microsoft.AspNet.SignalR.Core.dll
Both these SignalR vulnerabilities are false positives, they are about a very old bug in The Open Whisper Signal app before 2.23.2 for iOS, which is completely irrelevant to SignalR from Microsoft.
4. Microsoft.DiaSymReader.Native.x86.dll
5. Microsoft.DiaSymReader.Native.amd64.dll
Both these vulnerabilities are false positives, they suspect they have a vulnerability from 2006, that was already fixed in 2006. The vulnerability is matched to the Dia software by RedHat, which is drawing software inspired by Windows Visio.
6. YamlDotNet.dll
YamlDotNet was correctly identified as a dependency used by us, but the vulnerability coupled to it is only valid for version 4.3.2 and earlier. OWASP did not recognise that we are already using 6.1.1.

So we conclude that all security risks found by OWASP Dependency Check are false positives.

WhiteSource Bolt found 3 vulnerable dependencies:

1. microsoft.codedom.providers.dotnetcompilerplatform.████████.nupkg
2. System.Net.Http-████████.dll
3. jquery.████████.nupkg

Which equated to CVE-████████, CVE-████████, CVE-████████, CVE-████████ and CVE-████████. The first 4 are from Microsoft and have all been acknowledged and fixed in updated versions of our outdated packages. The jQuery is coupled to CVE-████████ and can be fixed with an update.

From this a conclusion can be made that WhiteSource Bolt found way less false positives, and OWASP did not find any more serious issues.

Appendix B

Appendix: Pipelines technical explanation

In the case study, the pipelines at SpendLab were discussed briefly and not in detail. To give some more technical insight, this appendix is added.

B.1 SpendLab Technology pipelines technical explanation

To understand what the QuickBuild.yml and BuildForRelease.yml exactly do, they are analysed below. First it is important to know the structure of the build server, when a pipeline is ran a folder is created with the following path:

```
D:\b1\w\"some number\"
```

The build server generates a number for this specific build, and then in this folder a Sources, Binaries, and Artifacts folder are created. In our Sources folder we get the full application folder, so the .git, APRA, wiki and more.

First they both use a template named Build-Framework-Solution.yml, this does the following:

1. Empty the Binaries directory
2. Copy the sources directory to the binary directory
3. Do a NuGet restore in the binaries directory, to make sure all dependencies are available before building the solution

4. Build solution in binaries directory, and save the finished build in Binaries/APRA_publish folder
5. Run tests from the built solution in the binaries directory

We empty the binary folder and copy the sources over, because the downloaded NuGet packages and intermediate build files by MSBuild should not be published. This would be a waste of bandwidth and time.

If a commit or pull request is submitted and a QuickBuild is done, only the above Build-Framework-Solution is executed.

When a BuildForRelease is done, some extra steps are done in addition to the Build-Framework-Solution:

1. Copy Sources to ArtifactStagingDirectory/s.
2. Run the Build-Framework-Solution.
3. Copy the created APRA_publish to the ArtifactStagingDirectory/r.
4. Zip the ArtifactStagingDirectory and send it to the approval stage.

B.2 Final pipeline definitions

Besides the changes described in section 5.3.2, there were some other more under the hood changes. The Solution is not build in the binaries directory anymore, the sources are directly copied to the ArtifactStagingDirectory before building, and the build is directly built to the ArtifactStagingDirectory. This makes a build for release significantly more efficient, because we do not have the inefficient copying of source files everywhere.

The changes made to the QuickBuild are as following:

- Added "trigger: none" to make sure this pipeline is never triggered, except as build policy on a pull request.
- Added steps to run SonarQube, these are only ran when the build reason is a pull request. If these steps fail, for instance if the SonarQube server is down, the build will not fail.
- Added a comment, as described in section 5.3.1 about SonarQube caveats, that shows a link to the SonarQube server.

And the general changes made to the build for release are:

- Clean the SourcesDirectory, this is standard not done to enable incremental builds and deployments. This is done because otherwise

old dependencies might still linger on the build server and clutter the WhiteSource Bolt report, as described in 5.4.1.

- Copy sources to ArtifactStagingDirectory.
- Build solution in SourcesDirectory with as output folder the ArtifactStagingDirectory.
- Added SonarQube steps, the same way as in QuickBuild.
- Added WhiteSource scan.

The DBQG uses exactly the same pipeline definitions for QuickBuild and BuildForRelease, but with different project name and parameters. They both use exactly the same Build-Framework-Solution, this is why the YAML files for the DBQG are not included.

B.2.1 APRA Quickbuild.yml

The full YAML specification has been redacted in this version.

B.2.2 APRA BuildForRelease.yml

The full YAML specification has been redacted in this version.

B.2.3 Build-Framework-Solution.yml

The full YAML specification has been redacted in this version.

Appendix C

Appendix: SonarQube setup

In the case study, it is not mentioned how SonarQube is set up in detail. In this appendix we will discuss some technical features of SonarQube, and how they were used for the case study at SpendLab.

C.1 Configuration SonarQube

It was decided to not use the SonarCloud subscription service that is provided by SonarSource, but to create our own SonarQube server. The server was set up in the following way:

1. Create an Azure VM, that is only accessible through the company network.
2. Configure and install SonarQube including an SQL database on this VM according to the documentation.
3. Set up a domain name, and configure SSL certificates through a reverse proxy.
4. Set up the projects within SonarQube, and make sure they are private. This is to force people to log in to access the source code.
5. Set up the pull request decoration for Azure DevOps from the SonarQube server by adding a *Personal Access Token* to SonarQube.

For now, SonarQube does not use personal accounts for developers. We created an Admin and Developer account, to make sure everyone can access the SonarQube server. SonarQube is able to easily integrate with existing

users using for instance OAuth, an open standard for access delegation. At the time of writing SpendLab did not use Azure Active Directory, which is the Microsoft implementation for identity management that also supports OAuth. Because of this, we could not easily expose the user-base to SonarQube.

C.2 Disabled rules in SonarQube

Written down in table C.1 are all SonarQube rules we disabled at SpendLab. No rules that have anything to do with security have been disabled, so this does not influence the research results. The amount of issues solved by disabling the rules is not entirely clear for older rules, because we found this metric only after disabling those rules. SonarQube did not log the amount of issues resolved by older disabled rules.

Language	Rule	Reason	Solved APRA	Solved DBQG
C#	Types should be named in PascalCase	This convention is not strictly used		
HTML	"<th>" tags should have "id" or "scope" attributes	Unnecessary for SpendLab	139	0
C#	"ISerializable" should be implemented correctly	Is ignored	26	17

Table C.1: SonarQube disabled rules

Appendix D

Appendix: Interviews at SpendLab

For the case study, some interviews were conducted. The two interviews and a summary of all the answers are presented here.

D.1 SonarQube interviews

First the questions asked for the SonarQube interview are enumerated. Then, for each interview, a summary is given. The interviews were conducted in Dutch, but the summaries given below are written down in English.

D.1.1 Questions

The bold keywords are used to make writing down the reviews later more compact and clear.

1. **Experience:** How long have you been working in the software development sector?
2. **Company experience:** How long have you been working at SpendLab?
3. **Role:** What is your current role at SpendLab?
4. **Encountered issues:** Have you ever encountered a security related problem found by SonarQube? If yes: can you elaborate?

5. **examples:** I will be giving three examples of SonarQube security bugs, in the context of APRA. For each example, can you explain if you agree with this issue? Would you solve it in the way SonarQube suggests or would you solve this another way?
 - (a) Handle the exception or explain in a comment why it can be ignored - Critical
 - (b) Refactor this code to not log tainted, user-controlled data. - Minor
 - (c) Make this field 'private' and encapsulate it in a 'public' property. - Minor
 - (d) Refactor this code to not perform redirects based on tainted, user-controlled data. - Blocker
6. **Usefulness comments:** The security comments made by SonarQube are all useful:
 - (a) Strongly disagree
 - (b) Disagree
 - (c) Neutral
 - (d) Agree
 - (e) Strongly agree
7. **Improvement:** Do you think SonarQube improves security at Spend-Lab?
8. **Opinion:** What do you think of SonarQube so far?
9. **Remarks**

D.1.2 Developer 1

Experience: 5.5 years.

Company experience: 1 year full time, 6 months internship.

Role: Software Engineer.

Encountered issues: Developer 1 did not get many security related issues from SonarQube, he would guess that he found around 8/10 comments useful. Around 20 percent of the rules, not necessarily security related, can occur a lot and he found these rules could be irritating.

Examples:

- (a) In first instance developer 1 would think this is a functional problem, he could not instantly see why this is a security issue. Maybe in certain context, he would solve it but why this is a security issue is not completely made clear by SonarQube. He would say this is indeed critical because it is incorrect error handling, but not because of the security risks associated.
- (b) Developer 1 agrees with minor, but does not directly see how this is security related.
- (c) Developer 1 did not see this as a vulnerability, but more as a code smell. So he does agree with the Minor label, but not exactly with that this should be marked as a vulnerability.
- (d) After reading the SonarQube explanation, the developer thinks this is a valid rule and a good solution from SonarQube. He also agreed with the blocker label.

Usefulness comments: Developer 1 Agrees with the statement that security comments made by SonarQube are all useful. He does not strongly agree because he feels like minor or info issues do not contribute that much to security.

Improvement: Developer 1 does not think he can say something about SonarQube improving the security, because he did not get security comments.

Opinions: Developer 1 sometimes does not like using SonarQube, because there are some rules he thinks are not necessary. He thinks this needs some tweaking, to make sure the pull request is not cluttered with rules deemed unnecessary by him or the team.

Remarks: No remarks.

D.1.3 Developer 2

Experience: 2 years part time.

Company experience: 1.5 years

Role: Software Engineer

Encountered issues: Developer 2 has installed SonarLint¹ in his IDE, so he sees some comments earlier. He does not see many security related problems from SonarQube, and SonarLint helps him find things like execution paths resulting in null earlier.

¹SonarLint is an IDE extension from the creators of SonarQube, that is able to show many of the SonarQube errors directly in the IDE while programming.

Examples:

- (a) Developer 2 thinks this rule is valid, but argues that critical in the context of security is maybe too high. When looking at code quality, he does think this rule is critical.
- (b) Developer 2 understands this rule, and can understand why it is only minor. With logging it could be intended and not really create any problems. In the end developer 2 thinks this is an useful rule.
- (c) Developer 2 thinks this should be changed, it does not impact functionality but it is about code quality. He does not think this is directly a vulnerability, but sees this as a code smell. He gets why it could be a vulnerability with custom get set code, but vulnerability might be a bit overblown.
- (d) Developer 2 thinks this is a valid rule, he agrees that SonarQube sees this rule as blocker.

Usefulness comments: Developer 2 strongly agrees with that comments made by the SAST are all useful. He has some experience with SonarQube and thinks they are a market leader with security.

Improvement: Developer 2 notes that SonarQube does not cover all security issues, but it is a gain for the company in security.

Opinions: Developer 2 thinks SonarQube is sometimes a bit frustrating to work with it, because he does a lot of small bug fixes. With many pull requests it can get a bit frustrating to wait a lot longer on a SonarQube scan, especially on some small changes probably not triggering errors.

Remarks: No further remarks.

D.1.4 Developer 3

Experience: 2.5 years, part time next to study.

Company experience: 2.5 years

Role: Junior Software Engineer.

Encountered issues: Developer 3 had some security related issues, especially when programming user management he had some comments related to sanitizing input. All these comments were taken serious by him.

Examples:

- (a) Developer 3 sees this rule as convenient, he does think that critical might be too high because it is most likely not a breaking issue.

- (b) Developer 3 understands this rule, and thinks in our case this is indeed minor. If the logs were used within a database or something this could be more dangerous. He also argues that you could enter many newlines or other clutter to make the logs unreadable.
- (c) This comment is useful according to developer 3, but he does not think this is specifically a vulnerability. He sees this more as a code smell, and does agree with the minor label.
- (d) Developer 3 argues that this is an useful comment, that is important to directly solve when it is found. He also agrees with severity label blocker.

Usefulness comments: Developer 3 strongly agrees with the statement made, he thinks that all comments are useful. He does note that he does not think all SonarQube comments marked as vulnerability are a direct vulnerability.

Improvement: Yes security is improved, because SonarQube definitely lets people think from a security point of view. It also opens the eyes to some things you would not think of.

Opinions: Sometimes it is inconvenient, especially on busy days with many developers and a lot of concurrent builds, SonarQube does hamper the development process. SonarQube is user-friendly according to developer 3, but he would like SonarQube to place all comments, because this is not the case as described in section 5.3.1.

Remarks: Developer 3 argues that the solution proposed in figure 5.4 to place a comment that not all comments by SonarQube are placed is the only comment we need. If SonarQube would place all comments that would be fine too, but now having it in two places is inconvenient.

D.1.5 Developer 4

Experience: 3 years.

Company experience: 3 years.

Role: Lead developer.

Encountered issues: Developer 4 also worked on the user management system, and some security related issues were found here. He also fixed a SonarQube comment which was a direct URL redirect, which was also fixed.

Examples:

- (a) Yes this should be solved according to developer 4, not specifically because he sees this as a security risk, but more because one big exception is not the code style from SpendLab. We need to create many more smaller exception and handle these exceptions separate. But developer 4 sees this more as a code smell. This being a security risk is context dependent according to developer 4.
- (b) Developer 4 sees this as an issue that is indeed minor, the potential impact he thinks is relatively small because it concerns only the logs.
- (c) The rule is true according to developer 4, and is convenient because it is easy to forget and sloppy programming. Strictly he thinks this works, but it might give unintended behaviour, so minor is the correct label.
- (d) This rule is useful according to developer 4, he does agree with it being a blocker rule.

Usefulness comments: Developer 4 thinks that he cannot answer this question, because he does not have an overview of everything SonarQube checks on. We rephrased the question to: SonarQube improves the security of applications developed at SpendLab. With this statement he strongly agreed.

Improvement: If it is about SonarQube against not using a tool at all, developer 4 thinks SonarQube definitely has an added value. Because even when we do not find vulnerabilities, it still does help to know that we searched for them. It also lets developers think about security, and lets them think about it on a basic level.

Opinions: According to developer 4, SonarQube is sometimes in the way, but this is more because we are new to using it. On the longer term, we know we are more secure because we are using SonarQube to check. It takes some initial effort to set up SonarQube, but we are not completely there yet. The impact on pipeline duration is big, but developer 4 agrees that this is not a problem because we get some security assurance in return.

Remarks: No further remarks.

D.1.6 Developer 5

Experience: 5 years.

Company experience: 3 years and 8 months.

Role: Lead developer.

Encountered issues: Developer 5 had one notable security issue, a hard coded password in the developer seed. The developer seed is used to create initial accounts to test for developers. This is strictly seen a security issue, but it is needed to test and not a risk later on.

Examples:

- (a) Developer 5 sees this as a good rule, but also does not know if this is specifically a security issue. This rule being a security issue is context dependant, but he notes that this is probably almost impossible for SonarQube to detect.
- (b) According to developer 5, this is a more severe security issue than the previous rule. So he does not agree with this being minor. For instance opening a log in a browser, you could maybe inject JavaScript code using this. In the SpendLab application this is no risk for now, but depending on the context this could be critical.
- (c) Developer 5 thinks this is minor, because in our context this would not really matter. A rule like this might give issues when you are using micro services, so it is indeed a security risk according to him. The minor label is thus also correct.
- (d) Developer 5 did not directly see the security issue when first seeing this rule, and would not directly see why this is a blocking issue using just the SonarQube explanation. He also argues that the first compliant solution is not good code, and is hard to maintain. So it is important to not blindly follow SonarQube suggestions.

Usefulness comments: Developer 5 agrees, he does not strongly agree because not all security risks are actually risks because they are context dependent. He does prefer that SonarQube comments on these issues, because then we think about these issues. He does not strongly agree because the vulnerability levels are not always correct according to developer 5.

Improvement: In the end, developer 5 does not really think that we significantly improve security using SonarQube. This is because our application is really closed, because we have a closed set of users that work within our company. This is based on trust and we can mitigate security risks by using the software in-house. SpendLab also uses frameworks that are known to be secure, and if they are not this would probably be notified by White-Source.

Opinions: Developer 5 thinks that in the end it is an improvement, but we do need to tune the rules more to our liking. It is also frustrating that not all comments are placed by SonarQube, as described in section 5.3.1.

Remarks: Developer 5 noted that he would rather have a little too many

issues found, and put them on wont fix, than not finding those issues by disabling rules or excluding files.

D.2 WhiteSource interview

For WhiteSource, only one interview was conducted because only developer 2 actually used the software. The following questions were asked, and the answers with them are thus given by developer 2.

1. Can you explain some of the issues found by WhiteSource:
WhiteSource found some major problems within our projects according to developer 2, which were definitely important to keep up to date. He also thought the outdated dependencies found by WhiteSource were useful.
2. What did you think of the interface of the tool:
Because we are using two solutions, APRA and DBQG, it is a bit complicated to see which vulnerable library is in which solution. There is also the problem that we only look at the DLLs. One of our DLLs had an issue in .NET Core, but we do not use this and it is hard to find where this issue comes from and how to solve it. The doubles described in section 5.3.1 were not a big issue according to developer 2, because it did show all the issues nonetheless.
3. WhiteSource is useful for security in our company:
Yes he does think that it is useful, because Azure DevOps does not do this automatically. He does not strongly agree because the usability of WhiteSource is holding it back.
4. Do you think WhiteSource improves security in our company:
Yes, if WhiteSource finds an issue it is a valid security issue that you have to patch. This is mostly indisputable according to developer 2, and issues found are thus useful.
5. Before using WhiteSource, were these vulnerabilities checked at Spend-Lab?
This was almost not done, developer 2 did find some issues with his private projects that use some of the same dependencies, because he uses GitHub that automatically finds issues without the need of a plugin like WhiteSource.
6. What do you in general think of using WhiteSource:
It is useful, but we need to streamline the workflow and have someone that knows this tool, and usability could definitely improve.

7. How many times would you think it is needed to seriously check and fix issues found by WhiteSource:

Minimal once a month, somebody has to check if we need to update dependencies. We should at least get the lists from WhiteSource and check if we should solve some of the issues found.

D.3 Extra interview questions

In the interview with developer 4, there was a new closed question formulated instead of the usefulness of comments. This question was deemed valuable to the research. Because section 5.5 about impact on productivity was later added, there were no questions about this in the original interview. To create a metric for this another question was added, the questions are as followed:

1. SonarQube improves the security of applications developed at Spend-Lab.
2. The extra time needed for building the pipelines is justified by the gain SonarQube gives in security.

Where the scale again goes from strongly disagree to strongly agree. In table D.1, the results are shown.

	Statement 1	Statement 2
Developer 1	Neutral	Strongly disagree
Developer 2	Agree	Neutral
Developer 3	Agree	Agree
Developer 4	Strongly agree	Agree
Developer 5	Neutral	Neutral

Table D.1: Results of extra interview questions

Developer 1 remarked that he did not personally see the security gains from SonarQube. He found the security rules from SonarQube useful, but they did not occur for him, so he strongly disagrees with statement 2.

Developer 2 noted that he thought neutral on statement 2, because he felt that there is room for optimization, and then he would agree more. He noted that using another stack might improve scanning time.

Developer 3 agreed with statement 2, but did not strongly agree. He felt that there might be an in between road, where we could for instance disable scans on smaller pull requests.

Developer 4 did not strongly agree with statement 2, because he also thought

that through configuration a lot of the downsides could be mitigated. If you look at it black and white, the time is worth it according to him, because it gives some peace of mind.

Developer 5 is neutral because he thinks SonarQube does work, but it did not find any big things yet that proved it's worth. He also noted that the issues found by SonarQube are not always issues in our context, because our application is used by a small user base in house. This way we mitigate many risks, with for instance IP white-listing.