Bachelor thesis
Computing Science

Radboud University

Security recommendations for Agile and DevOps development at Ridder Data Systems

Author:
Marc van de Werfhorst
s1009779

First supervisor/assessor:
dr.ir. Erik Poll
`E.Poll@cs.ru.nl`

Company supervisor:
Henk Schoemaker
`hschoemaker@ecisolutions.com`

Second assessor:
dr. Cynthia Kop
`C.Kop@cs.ru.nl`

June 28, 2020

Abstract

In this thesis, we look at the process of software development at Ridder Data Systems in Harderwijk. They do software development in an Agile and DevOps way. This way of development brings new challenges regarding security compared to a traditional waterfall model. The goal of this thesis is to research to get to know the best security practices in Agile and DevOps development. We are also going to look at the process at Ridder and focus on what they do with security. With this knowledge, we can then make recommendations for companies that make software in an Agile and DevOps way with regards to security and specifically for the process at Ridder.

Contents

Chapter 1

Introduction

In this thesis, we will take a closer look at improvements in the Agile and DevOps development process at Ridder Data Systems to improve the level of security. At Ridder, they make an ERP-system called Ridder iQ. ERP stands for Enterprise Resource Planning. The purpose of an ERP-system is to automate processes in a business to make the overall process more efficient.

Recently Ridder became part of a larger organization consisting of multiple similar companies like Ridder. These companies are now working towards a shared platform to exchange information between customers of these different companies. This new development led to concerns at Ridder since the information will be shared through the cloud. Security was not the focus when developing Ridder iQ. As long as the customer had a decent firewall, the data was safe. With the new shared platform, this is not the case anymore.

This leads to the following research question: What are the security practices that Ridder should be implementing? On top of this, there is an added layer of complexity. The layer of Agile and DevOps development. At Ridder, they have used Agile for some time now, while DevOps is still quite new. This kind of development brings new challenges compared to the traditional waterfall model. Therefore, we will be looking at general security practices and practices that are specific to Agile or DevOps.

To answer the research question, we will take a look at other literature to determine what the best security practices are in the respective fields. Then we will also look at the current process of developing at Ridder to see how security is currently implemented and where we can improve. These two parts then lead to recommendations specifically for Ridder and for software companies in general.

The thesis consists of the following chapters. Chapter 2 is a background on Agile and DevOps development and a comparison between the two. Chapter 3 will summarise security practices from literature ranging from general security practices to practices that are more specified for Agile and DevOps development. The current process of Ridder will be described in chapter 4. In chapter 5, we give the recommendations based on the literature and the process at Ridder. We will give suggestions for future work in chapter 6, and in chapter 7 will be the conclusions.

Chapter 2

Preliminaries: Agile and DevOps

In this chapter we describe the key concepts of Agile development in Section 2.1 and DevOps development in Section 2.2. Afterwards we compare these types of software development and look at their differences and similarities.

2.1 Agile

The term Agile software development is the product of a meeting in Snowbird, Utah, between 17 people in 2001. They were all advocates for different software development methodologies. The meeting was meant to exchange ideas and see if there are similarities in the way of thinking between the different methods. During the meeting, there were many things they did not agree on apart from a few things. Together they wrote the Agile Manifesto[2] on the main points that they did agree on and formed the Agile Alliance[3].

Agile development is not so much a way of working as it is a way of thinking. This is reflected in the Agile Manifesto. The Manifesto values individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan. These values are represented by the 12 principles of Agile software development. Working software as the primary measure of progress and early and continuous delivery are part of those principles.

The focus with Agile development is on the teams and the people that are part of that team. When the developers are enthusiastic about the product that they are building, they make a better product. Also, working in the same room and speaking face to face helps the process. When you put the right people together without giving them specific roles, they become a self-organizing team.

There are a lot of Agile development methodologies such as Scrum, Extreme Programming, and Feature Driven Development[3]. All these methodologies start with the values in the Manifesto. Every team must choose a methodology that fits with them and adjusts it according to their specific needs.

2.1.1 Scrum

Scrum is the methodology that is used at Ridder. Therefore we will take a closer look at this methodology.

Scrum was created by Ken Schwaber and Jeff Sutherland. Both are part of the Agile Alliance and co-created the Agile Manifesto in 2001. However, Scrum stems from the 1990s, and it was the reason they were at the meeting. Together they also wrote the Scrum Guide[24] in 2010 with the latest version coming from 2017. We use this Scrum Guide for this section since it is made by the creators of Scrum. Schwaber also founded Scrum.org[25] in 2009 to train people in the correct use of Scrum.

The three core values of Scrum are transparency, inspection, and adaptation. Transparency means that every member of the team knows what the goal is and how to achieve that goal. With frequent inspection, you ensure that you identify as early as possible if the product is going in the wrong direction. If it is going in the wrong direction, you can adapt the process to get the desired outcome.

With Scrum, the work is done in small groups of 5 to 11 people. One of those people is the Product Owner. The Product Owner is responsible for the Product Backlog. This is a list of specific tasks that need to be done and which tasks have the highest priority. Most of the people in the team are part of the Development Team. These are the people that do the work. Within the Development Team, there is no specified role, but it must be cross-functional. This means that all the necessary skills are present in the team. Another important aspect of the Development Team is that they are self-regulating. No one tells the team how they should tackle the different problems they encounter. The last role is that of the Scrum Master. The Scrum Master facilitates the team and guides them through the Scrum process.

The process of Scrum consists of multiple sprints. A sprint is a period of at most a month. During a sprint, there are four phases. The first phase is the Sprint Planning. In this phase, the team decides which features will be realized at the end of the sprint. This is done using the Product Backlog and the priority of the tasks on the Backlog. After this, there is a period in which the sprint is carried out. During this period, there is a Daily Scrum, which is a short meeting in which everyone tells the rest of the team what he or she is working on. At the end of the sprint, you have the Sprint Review. This is used to reflect on the product that came from the sprint. There is also a Sprint Retrospective for reflecting on the teamwork. This is to make sure that potential problems that arise during the sprint do not happen again.

2.2 DevOps

DevOps[11][28][33] is the combination of Development and Operations. Development is the place where the product is made, and the code is written. After Development is done and they have working code to be distributed to clients, it is handed over to Operations. Operations then deploys the product and handles everything to keep the product running. This includes handling issues raised by clients, for example. Traditionally these are two different departments in software companies. DevOps couples these departments more closely and lets them work together. A common visualization of this process is as in figure 2.1

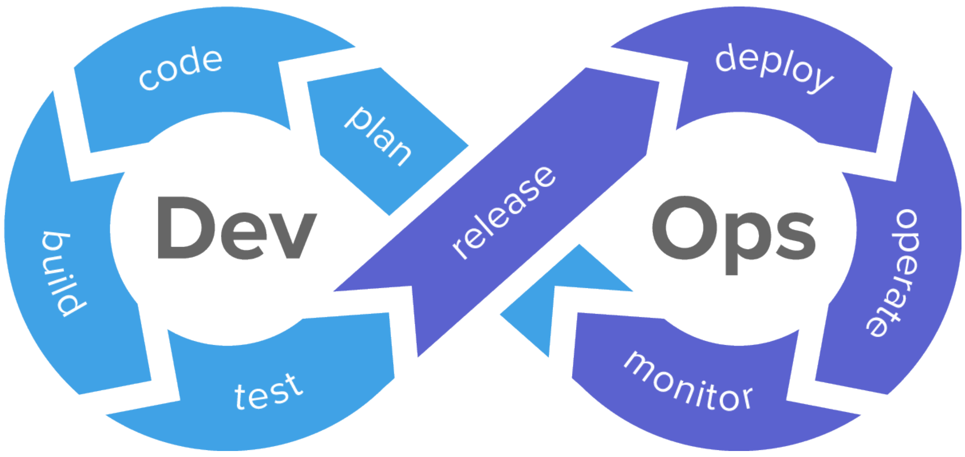

Figure 2.1: The DevOps cycle[13]

The book *DevOps: A Software Architect's Perspective*[6] gives a clear definition of DevOps. "DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production while ensuring high quality." With this definition, they also found the following five categories of practices that are DevOps related.

1. Operations should have a say in the making of the requirements. This coincides with the plan arrow in figure 2.1. Operations has to maintain the code after it is deployed, and having requirements regarding logging and monitoring can help with that.

2. Shorten the time between observation of an error and the fixing of that error. This involves making Development responsible for incident handling for the first period before it is handed over to Operations.

3. Make a standard for deployments and enforce those standards. This way, you avoid making errors when deploying, and it makes fixing errors easier. You have a clear history of the codebase and can easily see where it went wrong and what components most likely caused it.

4. Use a continuous deployment pipeline. This shortens the time between the commit of a developer and the deployment of code. Automated testing is often part of this pipeline, which ensures quality.

5. Treat infrastructure the same as application code. Scripts used for deployment should have the same standards as the code for the application. This minimizes the risk of a script being wrong and slowing down the deployment process.

Another important part of DevOps not mentioned in these categories is live testing and monitoring. This involves deploying new code and closely monitoring it. After the new code has passed all the tests, it can be added to the existing code. This ensures quality.

Using external tools is also a big part of DevOps. Most practices rely on some tool to do the task. The continuous deployment pipeline, monitoring, and testing all use different tools.

2.3 Comparison Agile and DevOps

Now that we know what Agile and DevOps are, we can compare them to each other. We will not go too deep into it since Ridder is using both.

The main difference[10] between the two is that Agile bridges the gap between development and the end-user, and DevOps bridges the gap between the different departments inside the company. The focus with Agile is on the development team, while with DevOps, the focus is on cooperation between different departments. For example, development and testing. Agile is all about getting a working product to show to the end-user as fast as possible so you can adapt to changing requirements. DevOps is about automating processes to deliver that product as fast as possible.

Despite these differences, the two are not mutually exclusive. If you want to have a more efficient process, the goal should be to do both. The core motivation for both Agile and DevOps is the fast delivery of a product, and that is why they work well together.

Chapter 3

Security practices

The purpose of this chapter is to explore what security practices are out there. We look at practices in general and specifically for practices that work in an Agile and DevOps environment. This forms the basis for the recommendations in chapter 5.

In this chapter, we look at three categories of sources for security practices. We divided the sources into three categories because the motivations behind the sources are different. In section 3.1, we talk about Maturity Models with the focus on the BSIMM Security Framework. They describe how security is handled in different companies. Section 3.2 is about Secure Software Lifecycle Processes, in particular Microsoft SDL. A lifecycle process is a list of practices that you should do without there being an explicit financial motive. Application Security Testing companies in the last category, and they are discussed in section 3.3. These AST companies do have a financial motive as they want to sell their products to increase security.

A lot of the sources discussed in this chapter are also mentioned in a paper by Williams at the Cyber Security Body of Knowledge(CyBOK) called the Secure Software Lifecycle Knowledge Area[32]. CyBOK is an organization, led by the University of Bristol, that create these knowledge areas to map established knowledge. In this case, the field of the Secure Software Lifecycle.

3.1 Maturity Models

Maturity models are the first of the three categories that we look at in this chapter. The purpose of a maturity model is for companies to compare to and see in which category they are behind. Often these maturity models are divided into domains. Such a domain holds a group of practices that fall in that domain. Each of these practices has different levels. These levels correspond to the level of maturity that a company has in that practice. When you go through the model and determine what your levels are for each practice, you have a baseline. With this baseline, you can see in which practice you lack, and you can plan a strategy for improving your security. Maturity models are also useful for getting ideas for ways to improve your security. This is also the reason for starting with maturity models because it gave a helpful checklist to get to know more about security at Ridder.

3.1.1 BSIMM Security Framework

The primary source we looked at is the BSIMM Framework[7], which stands for Building Security In Maturity Model. The BSIMM was made by experts at the Synopsys Software Integrity Group. Their idea was to gather data on Software Security Initiatives (SSIs) at other companies. From this data, they extracted which initiatives these companies had in common. These initiatives became the baseline of the first version of the model. They started with nine companies and have now analyzed over 100 companies. The goal of BSIMM is not to say how you should do security, but they show what others are doing for comparison to your own company.

After all the research at the different companies, they created the model. They took all the 119 initiatives and divided them into four domains. These domains are then split into three practices, and every practice has three levels. We will give a short overview of the different domains and practices below. The full list of domains, practices, and levels can be found in appendix A.

Governance

The first domain of Governance includes practices for organizing, managing, and measuring SSIs. Besides that is also the training of staff.

- **Strategy and metrics.** This practice is all about the division of roles and responsibilities within the company as well as identifying goals and setting up metrics to achieve those goals. It is also important to have everyone involved on the same page concerning security. Examples are assigning an evangelist for security or internally publishing your progress.
- **Compliance and policy.** Compliance and policy has to do with regulations from the outside world. For example, the GDPR from the EU and the AVG from the Dutch government. You should know what these regulations are and how you are going to comply with them. Examples of SSIs in this practice are combining all regulations in one internal document, which you then comply with or identifying where in the application the Personally Identifiable Information is.
- **Training.** The last practice of this domain is the training of software developers and architects. This includes awareness training for

everyone but also more specific training.

Intelligence

The Intelligence domain focuses on creating knowledge with regards to security.

- **Attack Models** The first practice of Intelligence is Attack Models and is a part of the knowledge that needs to be created in the company. You need to know what information you are processing and who your attackers could be.
- **Security Features and Design** The second practice concerns the security features in the product. The biggest part of this practice is that you should have separate modules for your security features, such as authentication. This way, you can focus on creating a good and rigid feature, and you do not have to make the same feature again for another part of the product with the risk of making mistakes. This should all be a single well-designed module or pattern.
- **Standards and Requirements** This last practice has a strong connection with the previous because this practice has initiatives for making standards and requirements for the features in the previous practice. Examples are to make security standards or a secure portal to exchange knowledge about security.

SSDL Touchpoints

This domain focuses on the analysis and quality assurance of all aspects of software development.

- **Architecture Analysis** Architecture analysis comprises capturing the product in clear diagrams and identifying risks and reviewing high-risk components, among other things.
- **Code Review** Code review encompasses the standard code review but also the use of analysis tools and reviews specifically for security features. Also, feedback on mistakes is included in this practice.
- **Security Testing** This practice contains initiatives for improving pre-release testing. These initiatives include adding security requirements into the test requirements and edge value testing.

Deployment

Deployment includes practices about external factors such as network settings and firewalls.

- **Penetration Testing** This practice consists of initiatives for both external and internal penetration testing. There is also a distinction between black-box and white-box testing.
- **Software Environment** The Software Environment practice is about the environment in which the product runs. This could be about reacting to updates of the operating system or monitoring the input that is given to the product.
- **Configuration Management and Vulnerability Management** The last practice concerns itself with patching applications and the tracking of bugs. Incidents and incident response is also part of this practice.

3.1.2 Other Maturity Models

Besides BSIMM, there are also other maturity models. We chose BSIMM as our maturity model because this model, together with the OpenSAMM model, is the most well known. We ended up with the BSIMM because this was the first that came to our attention.

Below is a list of other maturity models that might be useful to look at if you want more information. We did not look in detail at these other models.

- OWASP OpenSAMM[16]. The OpenSAMM is very similar to the BSIMM. It has the same structure of practices and levels as the BSIMM. The original OpenSAMM has four domains with three practices each. Currently, they are working on the second version, which has five domains with three practices each.
- OWASP DevSecOps Maturity Model[18]. The OWASP DevSecOps maturity model is the product of a German Master thesis by Timo Pagel. Because the thesis itself is written in German, and there is not much explanation on the English website of the project, we chose not to discuss this model in detail.
- NIST Cybersecurity Framework[14]. This is a framework made by the National Institute of Science and Technology commissioned by the US government to improve the critical infrastructure cybersecurity. We did not look closely at this framework, but because NIST is well-known in the field of computer science, we thought it was important to mention it.

3.2 Secure Software Lifecycle Processes

A secure software lifecycle process[32] is a combination of practices that say how you should do secure software development. It gives ways to tackle insecure software at the source rather than tackling security afterward. This is in contrast to a maturity model. Where a maturity model assesses what you already do, a lifecycle process tells you what to do to improve security.

3.2.1 Microsoft SDL and secure DevOps

Microsoft has its own lifecycle process called the Microsoft Security Development Lifecycle (SDL)[1]. It started as an internal development process and evolved into the SDL that it is today.

The 12 practices of Microsoft SDL

The Microsoft SDL consists of the following 12 practices.

1. **Provide Training.** For your product to be secure, everybody that works on it has to have some knowledge of security. Security is everyone's job. However, not everybody has to be a security expert.

2. **Define Security Requirements.** Next to functional requirements, you need to have security requirements. These security requirements also need to be dynamic, and they need to be updated when there is new information about threats. The optimal time to make these requirements is during the initial design or planning to minimize disruption. A good place to start would be the OWASP Top 10[17], for example.

3. **Define Metrics and Compliance Reporting.** When you have the security requirements, you need to check if you comply with those requirements. Everyone should be aware of the metrics, and code should pass the metrics before integrating it into existing code.

4. **Perform Threat Modeling.** Threat modeling consists of five steps. Define security requirements. Creating an application diagram. Identifying threats. Mitigating threats. Validating that threats have been mitigated. This technique should be routinely integrated into your development process.

5. **Establish Design Requirements.** When implementing a security feature, it must have precise requirements. A security feature is often complicated, and therefore mistakes can easily be made.

6. **Define and Use Cryptography Standards.** It is crucial to ensure data protection. Often cryptography is used to encrypt the data. Since

having bad crypto can have significant consequences, this should be left for experts. Therefore it is smart only to use encryption libraries that are verified by experts, and it should be easy to replace such a library.

7. **Manage the Security Risk of Using Third-Party Components.** You should have an inventory of all the Third-Party Components that you use. Besides that, you should be careful about which components you use and have a plan in case of a security breach in a component.

8. **Use Approved Tools.** Make a list of tools that have been cleared to use on the product, such as compilers. Strive to use the latest versions of those tools.

9. **Perform Static Application Security Testing (SAST)[1].** Use some static analysis tool before the compilation of the code to check for security vulnerabilities. Usually, this is a step in the commit pipeline but can also be part of the programming IDE.

10. **Perform Dynamic Application Security Testing (DAST)[2].** Similar to the previous practice, you should do some dynamic analysis. That is, with all components compiled and running as the end product would. The dynamic analysis then searches for possible vulnerabilities, memory corruption, or privilege escalation.

11. **Perform Penetration Testing.** An outside company specialized in penetration testing should do an analysis. They approach the product from the standpoint of a hacker and try to break it. This is the technique that finds the most vulnerabilities, but it can be more disruptive for the development process.

12. **Establish a Standard Incident Response Process.** A plan should exist to deal with security breaches. It is essential to know whom to contact and how you can mitigate risks when there is a breach.

8 practices for secure DevOps

Besides the SDL, Microsoft also has eight practices for secure DevOps. Some of the practices are also in the SDL, so for these, we will reference the SDL.

1. **Provide training.** See practice 1 of the SDL.

[1]Microsoft uses Analysis instead of Application as the A in SAST. However, it is more common to use Application, so we chose to use this for consistency.

[2]Microsoft uses Analysis instead of Application as the A in DAST. However, it is more common to use Application, so we chose to use this for consistency.

2. **Define Requirements.** On top of practice two from the SDL, you should integrate these requirements into your pipeline.

3. **Define Metrics and Compliance Reporting.** See practice 3 of the SDL.

4. **Use Software Composition Analysis (SCA) and Governance.** This is similar to practice 7 of the SDL. SCA tools can scan these third-party components and report any known vulnerabilities.

5. **Perform Threat Modeling.** See practice 4 of the SDL. While threat modeling can be seen as slow, it is still a good practice to do even if it is only used in the places that have the most risks.

6. **Use Tools and Automation.** This is a combination of practice 8, 9, and 10 of the SDL with the added remark that you vet your tools before using. Tools should integrate into the pipeline, not require security expertise, and avoid false-positive results.

7. **Keep Credentials Safe.** Do not store credentials or other sensitive information in the source files.

8. **Use Continuous Learning and Monitoring.** Monitor your product during the development and after deployment. This way, you can spot security vulnerabilities early and fix them. It can also help identify and contain an attack if it occurs.

3.2.2 Other Lifecycle Processes

There are several other lifecycle processes besides Microsoft SDL, which we discuss in this section. We will not go into much detail here, but it can be interesting to compare these to the Microsoft SDL.

The reason we chose the Microsoft SDL as our lifecycle process is partly because it is made and used by Microsoft. Microsoft is a big, well-known, and successful software company. Besides that, Microsoft SDL was one of the first lifecycle processes, and it is widely accepted in the industry.

SAFECode

The Software Assurance Forum for Excellence in Code (SAFECode)[19] is an organization consisting of members of the industry, such as Adobe, Microsoft, and Google. SAFECode is dedicated to promoting security practices to increase security in software, hardware, and services.

SAFECode publishes the *Fundamental Practices for Secure Software Development*[22], of which currently, the third edition is the newest. In this document, they summarise a lot of the practices used by the members of SAFECode.

Besides this document, they also published other, more specific papers. This ranges from a paper on security training[20] to a guide for Agile practitioners[21]. They also started an interesting new series, of which at the time of writing, only the first part is published, called *The Six Pillars of DevSecOps*.

Touchpoints

The seven touchpoints for software security are made by Gary McGraw. He is well-known in the field of secure software development and is still active in this field. He was also a co-author of the first version of the BSIMM in section 3.1.1.

The first time these touchpoints were introduced is in an article in the IEEE Security & Privacy Magazine[4]. After this, they appeared in a book written by McGraw called *Software Security: Building Security In*[9]. Since they are quite old and not as well maintained as the Microsoft SDL, we briefly mention the seven touchpoints, but we will not go further into them. The touchpoints are sorted in order of effectiveness in the experience of McGraw.

1. Code review
2. Architectural risk analysis
3. Penetration testing
4. Risk-based security tests
5. Abuse cases
6. Security requirements
7. Security operations

3.3 Application Security Testing Companies

In this section, we will discuss practices that are given by companies that make application security testing tools. These companies make and sell software to help with increasing the security of your project, so they also have some secondary motives to make security necessary. Nonetheless, it is essential to look at their recommendations since they have much experience in improving security.

3.3.1 Synopsys

One of the reasons Synopsys[26] is on our list of security companies is that they are a leader in the Gartner Magic Quadrant for Application Security Testing[8]. Gartner is a major advisory company that provides insight to decision-makers to achieve their goals. The Magic Quadrant for AST is for decision-makers to decide which security company they want to do business with. Synopsys is a leader in this Magic Quadrant, which means that their product is pervasive and easy to implement. Besides Synopsys being a leader, they are also the company behind the BSIMM Framework in section 3.1.1. This shows that they are also concerned with improving software security in general.

Synopsys has six main points on which you should focus to increase security[27].

1. **Integrate security into your DevOps environment.** This means that you need to integrate and automate security tools in your pipeline. It ensures that all the necessary security checks are always run.
2. **Build a holistic AppSec program across your organization.** Make sure that your people and technology are all up to date to defend against threats properly. This includes training employees.
3. **Get on-demand security testing for any application.** This practice says that you should hire the experts at Synopsys to do security testing.
4. **Find and fix quality and compliance issues early in development.** By doing this, you increase the reliability of your product, and you avoid extensive maintenance at a later point. Besides this, you make sure that you comply with the relevant regulations.
5. **Identify open source, code quality, and security risks during mergers and acquisitions.** This is relevant when buying other companies or merging with them that make software. You need to know what the vulnerabilities are and how you can respond to them.
6. **Assess your AppSec threats, risks, and dependencies.** Besides security testing, you should think of possible attack scenarios and know what the targets can be for these attacks. This is needed to avoid security breaches.

3.3.2 Veracode

Veracode[31] is just like Synopsys, a leader in the Gartner Magic Quadrant for Application Security Testing[8]. They are also a member of SAFECode, which is a non-profit organization that is dedicated to improving software security, as described in section 3.2.2. This shows that Veracode is invested in increasing security in general instead of only making a profit.

To increase security Veracode has the *5 Essential Steps For Shifting Security Left*[30]. This means that the security is considered as early in the development lifecycle as possible.

1. **Autonomous security from day 1.** This means that you should automate security checks from the beginning of the project. This way, the necessary security checks are always done without requiring a developer or an outside team to look at them.
2. **Integrate as you code.** You should integrate Application Security Testing as early as possible in the development lifecycle. This provides a tight feedback loop to fix possible problems.
3. **Avoid false alarms.** A false alarm is a notification that there is a security vulnerability when there is not. You want to avoid these false alarms to minimize the impact of them on the development process.
4. **Create security champions.** In every Scrum-team, there should be a security champion. This person is one of the developers on the team. By giving this person extra training, they become the security expert for the team. This way, there is always someone present at the meetings to keep an eye on security.
5. **Develop a culture of visibility.** This builds on the DevOps philosophy of developers being responsible for live running code. They are also responsible for the security of the same code. They need to be able to monitor the security so they can see when something is going wrong. They also need to know where the critical points in the system are.

3.3.3 SonarQube

SonarQube is a static analysis tool by SonarSource. SonarQube is used in development pipelines to find bugs, code smells, and security vulnerabilities. Code smells are pieces of code that do not necessarily contain bugs but are not up to the standard and are messy or inconsistent. SonarQube finds these problems by analyzing the source code and compare that source code to the rules that are activated. We chose to include SonarQube in this section because it is one of the main tools that are used at Ridder, as we will see in section 4.2.

There is a thesis[29] by Jesse van Son in which he uses SonarQube. The thesis is a case study to increase security in DevOps pipelines. To increase the security, Jesse used SonarQube for Static Application Security Testing by enabling additional rules to check for security vulnerabilities.

The outcome of the research was that enabling these extra rules worked, and it increased the security of the product. The downside to using the tool is that it takes a long time to set up and configure properly.

Chapter 4

Current process at Ridder

The information described in this chapter was gathered by three meetings of roughly two hours, each with a contact at Ridder. Our contact was the Technical Product Manager for their product. During these meetings, we posed a series of questions about the process at Ridder about the Agile and DevOps methodologies described in chapter 2. The answers to these questions were the basis for section 4.1 about Agile and section 4.2 about DevOps. We also used the BSIMM Framework from section 3.1.1 to establish the level of security at Ridder in section 4.3.

With the information in this chapter together with the practices from chapter 3, we can make recommendations specific for Ridder in chapter 5.

4.1 Agile at Ridder

At Ridder, the development department is split up in 4 teams of 6 people. These teams consist of 4 developers, a tester, and a Product Owner. Three of the teams work on bigger projects, such as a new module, and the other team tackles the smaller problems that can be done in a couple of days.

The teams work in sprints of two weeks. For the product backlog of these sprints, the team gets guidelines from product management. They get their ideas from two places. They either look at new developments on the market, or they get suggestions from their clients.

Ridder does not have a stringent definition of done. However, there are a few things that need to happen on every piece of new code. It has to be unit tested, but there is no minimum percentage of code coverage. Besides this, it has to be tested by a tester and accepted by the product owner of the team.

Testers and Product Owners from all teams meet regularly. These meetings are to discuss new developments inside and outside the company and to keep everyone on the same page. For example, testers need to make sure that they all test the same way and keep to the same standards. They could also discuss new ways of testing if they become available. Product Owners meet to keep the product as a whole consistent.

4.2 DevOps at Ridder

At Ridder, there is not a distinction between a Development and an Operations department. In general, everyone is responsible for their code. Most of the process to release is automated. However, there is one person that has the task to oversee Operations. He makes sure that everything keeps working and he takes care of possible alerts that come from the monitoring software Datadog.

After a piece of code is written, it begins its journey through the pipeline. All features are developed on a separate feature branch in Gitlab. The first phase of the pipeline is unit tests and SonarQube. Unit tests run one or several test cases for small pieces of code. SonarQube is a static analysis tool that can check the code for bugs, code smells, and security vulnerabilities based on rules described in section 3.3.3. Ridder uses rules to recognize security vulnerabilities that are in the OWASP Top 10[17] and the SANS Top 25[23]. You can find the full list of security rules in appendix C. After this phase of automated testing, both SAST and unit testing, a review of another developer is needed.

When the new code passes all the previous steps, it is sent to the tester. The tester will test the new code using their testing tools and perform unit tests and sanity tests. Sanity tests exist to make sure the product has reasonable behavior before running the more costly tests. The tester will also create new automated unit tests. These automated unit tests will be added to the existing tests that run during the night. Automated tests ensure that everything still works after adding new code.

After a sprint of two weeks, the new functionality will be released for the part of the product that runs in the cloud. The part of the product that runs on the customer's server has a release every two months. The customer has to update the software on their server themselves manually. During this update, the system is down, and the work cannot continue. That is why Ridder has chosen to release every two months instead of two weeks. For both the release to the cloud and the customer, it is just a matter of pushing a button without extra input needed from the developers.

4.3 BSIMM at Ridder

To better understand what level of security there is at Ridder, we filled in the checklist from the BSIMM Framework in section 3.1.1. The conclusion that can be made from this is that Ridder does some things right, but not in a structured way. Due to this, most of the security practices mentioned in chapter 3 can have a positive impact on the level of security at Ridder, so the security practices did not narrow down that much when selecting the recommendations in chapter 5. You can find the filled-in checklist for Ridder for level 1 in appendix A.

When going over the checklist, we quickly noticed that there is still much work to be done to reach level 1 in most of the practices. Because of this and because it was very time consuming, we only discussed initiatives from the first level of the practices. It took an hour and a half to discuss these initiatives. For a full evaluation of the BSIMM at Ridder, you should do all the levels, since you reach a higher level if you do only one of the initiatives of that higher level.

Doing this checklist was very useful for establishing the baseline at Ridder. Another benefit of doing the BSIMM checklist was a raised awareness of security. Our contact at Ridder stated that he was much more aware of security after seeing what you can do to increase security.

During our meetings, we started with filling in the BSIMM Framework, and after that, we discussed their Agile and DevOps process. Next time we would do this the other way around. It is easier to understand the level of security when you know what the development process is.

Chapter 5

Recommendations

This chapter takes the security practices from chapter 3 and the description of the process at Ridder in chapter 4 to form recommendations. First, we make recommendations for software development companies in general. After this, we make some additional specific recommendations for the process at Ridder. We conclude the chapter with a section with some additional tips.

5.1 General recommendations

In this section, we give recommendations for software companies in general that use Agile and DevOps and have about the same level of security as Ridder. We came to these recommendations by looking at the security practices in chapter 3 and comparing the different sets of security practices in that chapter. Most of the recommendations in this chapter are part of all or most of the sets of security practices and thus deemed them essential. The selection of recommendations was also based on the level of security at Ridder, which was very basic, as explained in section 4.3. Furthermore, we chose recommendations that fit well at Ridder and are easy to implement there.

We divided the recommendations in this section into three parts. Recommendations that fit in the Agile methodology, recommendations that fit in the DevOps methodology, and general security recommendations. The recommendations in the first two parts can also work in other types of software development but are easier to implement when used in their respective methodologies.

5.1.1 Agile recommendations

Here we give recommendations that fit in the Agile methodology. The recommendations can work in other types of software development but are easier or more intuitive to implement when Agile is used.

- **Define security requirements.** When creating the backlog for the project or a sprint, there should be room for security requirements. These requirements should have the same importance as functional requirements.

  Almost all of the sources in chapter 3 contain some form of this recommendation. When explicitly defining these requirements, you are forced to think about security, and you make time free in sprints to work on them. It is not an afterthought. These are the reasons why it is important.

- **Create security champions in every team.** These security champions are developers that are part of a scrum-team. The security champion is the security expert in a scrum-team. Other developers in the team can come to them with questions about security. This way, you also have someone in the meetings that keeps an eye on security.

  Veracode's *5 Essential Steps For Shifting Security Left* from section 3.3.2 is the main source for this recommendation, because they used the term security champion. The BSIMM Framework from section 3.1.1 also mentions a form of this. We think it is important, because of the input the security champion can give during team meetings. The security champion also has the responsibility to make sure the security is up to the standards.

5.1.2 DevOps recommendations

Here we give recommendations that fit in the DevOps methodology. The recommendations can work in other types of software development but are easier or more intuitive to implement when DevOps is used.

- **Integrate security testing in your development pipeline.** By security testing, we mean the use of Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) tools, but also other types of testing like unit testing with security test cases. You can do this by implementing SAST tools or the OWASP package checker[15], for example.

  The use of tools for security testing is part of all the sources in chapter 3. Once the tools are set up, you do not have to look at them. It does not require manual labor. They do their job and hopefully help you make a safer product.

5.1.3 General recommendations

Here we give general recommendations that are not specific for Agile or DevOps. These recommendations can work for any type of software development.

- **Training of employees to create a basic understanding of security in the whole organization.** Developers should get training to be aware of security and how they can make the product more secure. Not only should the developers get training, but also management should get training to help them understand why you need to spend time on security.

  All of the sources that we looked at had some sort of security training for developers. Besides that, BSIMM, Microsoft SDL, and Synopsys state that everybody in the organization should have some basic understanding of security.

- **Know where your weak spots are.** You need to identify where your security is weak. This could be open-source software that you use or places where user data is being processed.

  This is important because when you know where your weak spots are, you can take action to reduce the risk of those weak spots. This recommendation is also part of most of the sources in chapter 3.

- **Perform penetration testing.** This is one of the hardest recommendations to do in an Agile and DevOps environment because the product is not static. However, it is part of most of the sources from chapter 3, and we think it is still valuable to get an outside perspective on your security by security experts. Large parts of the code are not regularly touched, so vulnerabilities found in those parts are always valuable.

- **Fill in the BSIMM Framework to improve awareness.** This is a recommendation that came from the meetings with Ridder. During one of the meetings, we discussed the BSIMM Framework, as discussed in section 4.3. After the meeting, our contact stated that he was much more aware of the possible ways to increase security. We only discussed the level 1 initiatives, but for a full evaluation of the BSIMM, you should do levels 2 and 3 as well.

5.2 Specific recommendations for Ridder

In this section, we discuss some additional recommendations that are specific for Ridder on top of the recommendations in the previous section.

- **Expand SonarQube rules.** This recommendation comes from a thesis by Jesse van Son discussed in section 3.3.3. He found that using Sonarqube, the security of the product improved. Since SonarQube is already used at Ridder for code quality and some security, it can be useful to look for some additional rules for security. There are external plug-ins that you can use in SonarQube to get more rules. These rules are added easily so that it could be a quick win.

- **Security champions meeting regularly.** We saw in chapter 4 that the testers and the Product Owners from all teams meet regularly. This could be expanded to the security champions from the recommendation in section 5.1.1. This is also part of the practice from Veracode, only they let them meet with the security team. Since there is no dedicated security team at Ridder, the security champions can form the security team in this way. They could coordinate on how to handle specific security requirements and keep up to date with new attacks.

5.3 Additional Tips

In this section, we give some additional tips that can be useful when implementing the recommendations from the previous sections.

Microsoft has some more information about its SDL practices on their website[1]. This includes some of the tools with which you can put the SDL into practice. It also includes other useful links, such as recommendations for cryptographic protocols.

Chapter 6

Future Work

This chapter is for ideas that rose during the research but was outside the scope of this thesis or too big for this thesis.

ISO standards

An interesting angle for increasing security in a company can be to look at the ISO standards for information security management. These are published by the International Organization for Standardization. These are the standards from the 27000 family. You can get a certificate to show that you comply with those standards. It can be useful to look at these standards and see what the requirements are to help plan possible improvements in security.

BSIMM at other companies

As mentioned in the introduction of this thesis, Ridder is part of a bigger organization consisting of other companies like Ridder. It can be interesting to look at these other companies and fill in the BSIMM Framework for them. With this knowledge, you can compare the different companies, and they can help each other. If one company is very mature in a practice, then it can help the other companies become more mature by sharing what they do.

OpenSAMM and SAFECode

In chapter 3, we looked at BSIMM for lifecycle processes and at Microsoft SDL for maturity models. However, OpenSAMM and SAFECode for those categories, respectively, look also very promising. We already gave a short description in chapter 3, but it could be interesting to look more closely at these sources and perhaps compare them to the ones we used. We did not have the time to do this ourselves.

Chapter 7

Conclusions

In this chapter, we will summarise the research and give conclusions.

7.1 Agile and DevOps

As we have seen in chapter 2, Agile and DevOps are both excellent ways to have a more efficient development process, and they work well together. Both Agile and DevOps require a change in the way of thinking when adapting this in your development process.

Security with Agile and DevOps is harder in some ways and easier in other ways compared to the traditional waterfall model. It is harder because you do not have a dedicated period after the product is done to take care of security. Things like penetration testing are more complicated because the product is always evolving. It is easier because automating security is more intuitive than in the waterfall model. When integrated into the development pipeline, there is no manual labor required. You can spot security flaws earlier and solve it correctly instead of plugging it after the product is done.

7.2 Sources for Security Practices

During this thesis, we identified three types of sources for security practices: maturity models, secure software lifecycle processes, and companies that make application security testing tools. Maturity models assess the current level of security, while lifecycle processes and security companies give guidelines for what you should do to increase security. You can find these categories and their examples in chapter 3.

We noticed that a lot of the practices we found in the sources either assume some level Agile or DevOps already, or they steer you in that direction. Especially the application security testing companies. For example, automating security testing like running SAST and DAST tools and automated unit tests is typical for a DevOps process. DevOps is all about automating to avoid manual labor.

Another thing we noticed while writing this thesis is the lack of traditional scientific sources like publications in scientific journals on the topic of security practices. There are plenty of scientific sources for a single security practice[5], but not that much on a set of security practices. The ones we found[12] for the sets of security practices reference the sources like Microsoft SDL that we used in this thesis.

7.3 Security Recommendations

During the project, we created security recommendations for Ridder Data Systems. Below we list these recommendations. For a more detailed overview of these recommendations, we refer to chapter 5.

- Define security requirements.
- Create security champions in every team.
- Integrate security testing in your development pipeline.
- Training of employees to create a basic understanding of security in the whole organization.
- Know where your weak spots are.
- Perform penetration testing.
- Fill in the BSIMM Framework to improve awareness.

7.4 Reflection on the Process

We had a decent start to the project with many sources to look at. This resulted in a long list of possibly interesting sources which we went through one by one. If we found a useful source, we would start by taking a closer look at that source. Next time, we should put that source aside and first look at all the other sources. We should do a quick scan through all the documents and look for the credentials of the writer to determine if a source is useful. That way, we know earlier which sources are useful ones, and we can manage our time better. There are some sources we did not get to take a closer look at because of this. There could also be more sources out there that are useful, but we did not find them because of this.

Another thing I should do differently next time is to start writing as soon as possible. I noticed that getting words on paper is not my best quality. The stress that this causes at the end of the project can be avoided by forcing myself to start writing earlier.

Bibliography

[1] Microsoft Security Development Lifecycle. https://www.microsoft.com/en-us/securityengineering/sdl/.

[2] Agile Alliance. Manifesto for Agile Software Development. https://agilemanifesto.org/.

[3] Agile Alliance. What is Agile Software Development? https://www.agilealliance.org/agile101/.

[4] Z. Azham, I. Ghani, and N. Ithnin. Security backlog in scrum security practices. In *2011 Malaysian Conference in Software Engineering*, pages 414–417, 2011.

[5] Z. Azham, I. Ghani, and N. Ithnin. Security backlog in scrum security practices. In *2011 Malaysian Conference in Software Engineering*, pages 414–417, 2011.

[6] Len Bass, Ingo M. Weber, and Liming Zhu. *DevOps: a software architects perspective.* Addison-Wesley, 2015.

[7] BSIMM. Building Security In Maturity Model. https://www.bsimm.com/.

[8] Gartner. Magic Quadrant for Application Security Testing, 2020. https://www.gartner.com/doc/reprints?id=1-1YWZKUB5&ct=200429&st=sb%20.

[9] Gary McGraw. *Software Security: Building Security In.* Addison-Wesley, 2006.

[10] Microsoft. DevOps vs. Agile. https://azure.microsoft.com/en-us/overview/devops-vs-agile/.

[11] Microsoft. What is DevOps? https://azure.microsoft.com/en-us/overview/what-is-devops/.

[12] P. Morrison. A security practices evaluation framework. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 935–938, 2015.

[13] Nub8. What is DevOps? https://nub8.net/what-is-devops/.

[14] National Institute of Standards and Technology. *Framework for Improving Critical Infrastructure Cybersecurity version 1.1.* 2018.

[15] OWASP. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/.

[16] OWASP. Software Assurance Maturity Model (SAMM). https://owaspsamm.org/.

[17] OWASP. Top Ten Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[18] Timo Pagel. OWASP Devsecops Maturity Model. https://owasp.org/www-project-devsecops-maturity-model/.

[19] SAFECode. SAFECode — Driving Security and Integrity. https://safecode.org/.

[20] SAFECode. Security Engineering Training, 2009. http://safecode.org/wp-content/uploads/2018/01/SAFECode_Training0409.pdf.

[21] SAFECode. Guidance for Agile Practitioners, 2012. http://safecode.org/wp-content/uploads/2018/01/SAFECode_Agile_Dev_Security0712.pdf.

[22] SAFECode. Fundamental Practices for Secure Software Development, 2018. https://safecode.org/wp-content/uploads/2018/03/SAFECode_Fundamental_Practices_for_Secure_Software_Development_March_2018.pdf.

[23] SANS. CWE/SANS TOP 25 Most Dangerous Software Errors. https://www.sans.org/top25-software-errors.

[24] Ken Schwaber and Jeff Sutherland. The Scrum Guide, 2017. https://www.scrumguides.org/scrum-guide.html.

[25] Scrum.org. The Home of Scrum. https://www.scrum.org/.

[26] Synopsys. Synopsys — EDA Tools, Semiconductor IP and Application Security Solutions. https://www.synopsys.com/.

[27] Synopsys. Synopsys Security — Software Integrity Group. https://www.synopsys.com/software-integrity.html.

[28] Synopsys. What Is DevOps and How Does It Work? https://www.synopsys.com/glossary/what-is-devops.html.

[29] Jesse van Son. Security by design in Azure DevOps pipelines, a case study at SpendLab technology, Bachelor's thesis, Radboud University, 2020.

[30] Veracode. 5 Essential Steps for Shifting Security Left. `https://www.veracode.com/resources/video/five-essential-steps-shift-security-left`.

[31] Veracode. Use Veracode to secure the applications you build, buy & manage. `https://www.veracode.com`.

[32] Laurie Williams. *The Cyber Security Body of Knowledge*, chapter Secure Software Lifecycle. University of Bristol, 2019. Version 1.0.

[33] L. Zhu, L. Bass, and G. Champlin-Scharff. DevOps and Its Practices. *IEEE Software*, 33(3):32–34, 2016.

Appendix A

BSIMM checklist

In this appendix, we list all the Software Security Initiatives per level per practice per domain of the BSIMM Framework.

A.1 Governance

A.1.1 Strategy and metrics

Level 1

- Publish process and evolve as necessary. Everyone knows the plan and is on the same page.
- Create evangelism role and perform internal marketing. Pick someone to help the teams build more secure software by talking with them.
- Educate executives to convince them to take security seriously.
- Identify gate locations, gather necessary artifacts. Determine what level of security should be required when releasing without enforcing it.

Level 2

- Publish data about software security internally.
- Enforce gates with measurements and track exceptions.
- Create or grow a satellite.
- Require security sign-off.

Level 3

- Use an internal tracking application with portfolio view.
- Run an external marketing program.
- Identify metrics and use them to drive budgets.
- Integrate software-defined lifecycle governance.

A.1.2 Compliance and policy

Level 1

- Unify regulatory pressures. Combine all different regulations into one set of things to comply to.
- Identify Personally Identifiable Information obligations.
- Create policy to handle with regulations. Everyone should comply to the policy and don't have to keep checking all regulations.

Level 2

- Identify PII inventory.
- Require security sign-off for compliance-related risk.
- Implement and track controls for compliance.
- Include software security SLAs in all vendor contracts.
- Ensure executive awareness of compliance and privacy obligations.

Level 3

- Create a regulator compliance story.
- Impose policy on vendors.
- Drive feedback from software lifecycle data back to policy.

A.1.3 Training

Level 1

- Conduct awareness training.
- Deliver role-specific advanced curriculum.
- Deliver on-demand individual training.

Level 2

- Enhance satellite through training and events.
- Include security resources in onboarding.
- Create and use material specific to company history.

Level 3

- Reward progression through curriculum.
- Provide training for vendors or outsourced workers.
- Host software security events.
- Require an annual refresher.
- Establish SSG office hours.
- Identify new satellite members through training.

A.2 Intelligence

A.2.1 Attack Models

Level 1

- Create a data classification scheme and inventory to see which applications use the most data or most sensitive data. The higher classification uses more data and has a higher priority.
- Identify potential attackers.
- Gather and use attack intelligence. Keep up to date with new attack models.

Level 2

- Build attack patterns and abuse cases tied to potential attackers.
- Create technology-specific attack patterns.
- Build and maintain a top N possible attacks list.
- Collect and publish attack stories.
- Build an internal forum to discuss attacks.

Level 3

- Have a science team that develops new attack methods.
- Create and use automation to mimic attackers.
- Monitor automated asset creation.

A.2.2 Security Features and Design

Level 1

- Build and publish security features. Not every team needs to make the same features for authentication for example.
- Engage the Software Security Group with architecture teams. Have someone from the SSG present at architecture meetings.

Level 2

- Leverage secure-by-design middleware frameworks and common libraries.
- Create an SSG capability to solve difficult design problems.

Level 3

- Form a review board or central committee to approve and maintain secure design patterns.
- Require use of approved security features and frameworks.
- Find and publish mature design patterns from the organization.

A.2.3 Standards and Requirements

Level 1

- Create security standards.
- Create a security portal that is easily accessible for everyone. Here you have all the information regarding security, possibly a wiki page maintained by the SSG.
- Translate compliance constraints to requirements.

Level 2

- Create a standards review board.
- Identify open source.
- Create SLA boilerplate.

Level 3

- Control open source risk.
- Communicate standards to vendors.
- Use secure coding standards.
- Create standards for technology stacks.

A.3 SSDL Touchpoints

A.3.1 Architecture Analysis

Level 1

- Perform security feature review.
- Perform design review for high-risk applications.
- Have SSG lead design review efforts.
- Use a risk questionnaire to rank risks of applications.

Level 2

- Define and use AA process.
- Standardize architectural descriptions.

Level 3

- Have engineering teams lead AA process.
- Drive analysis results into standard architecture patterns.
- Make the SSG available as an AA resource or mentor.

A.3.2 Code Review

Level 1

- Have the SSG perform ad hoc review.
- Use automated tools along with manual review.
- Make code review mandatory for all projects.
- Use centralized reporting to close the knowledge loop and drive training. Bugs found during code review are stored in a central database so everyone can learn from them.

Level 2

- Assign tool mentors.
- Use automated tools with tailored rules.
- Use a top N bugs list (real data preferred).

Level 3

- Build a capability to combine assessment results.
- Eradicate specific bugs from the entire codebase.
- Automate malicious code detection.
- Enforce coding standards.

A.3.3 Security Testing

Level 1

- Ensure QA supports edge/boundary value condition testing.
- Drive tests with security requirements and security features. For example checking that a user is timed out after repeatedly giving the wrong password.

Level 2

- Integrate black-box security tools into the QA process.
- Share security results with QA.
- Include security tests in QA automation.
- Perform fuzz testing customized to application APIs.

Level 3

- Drive tests with risk analysis results.
- Leverage coverage analysis.
- Begin to build and apply adversarial security tests (abuse cases).

A.4 Deployment

A.4.1 Penetration Testing

Level 1

- Use external penetration testers to find problems.
- Feed results to the defect management and mitigation system.
- Use penetration testing tools internally.

Level 2

- Penetration testers use all available information.
- Schedule periodic penetration tests for application coverage.

Level 3

- Use external penetration testers to perform deep-dive analysis.
- Have the SSG customize penetration testing tools and scripts.

A.4.2 Software Environment

Level 1

- Use application input monitoring to see what is given as input when the software is released to spot possible attacks.
- Ensure host and network security basics are in place.

Level 2

- Publish installation guides.
- Use code signing.

Level 3

- Use code protection.
- Use application behavior monitoring and diagnostics.
- Use application containers.
- Use orchestration for containers and virtualized environments.
- Enhance application inventory with operations bill of materials.
- Ensure cloud security basics.

A.4.3 Configuration Management and Vulnerability Management

Level 1

- Create or interface with incident response to react to incidents.
- Identify software defects found in operations monitoring and feed them back to development.

Level 2

- Have emergency codebase response.
- Track software bugs found in operations through the fix process.
- Develop an operations inventory of applications.

Level 3

- Fix all occurrences of software bugs found in operations.
- Enhance the SSDL to prevent software bugs found in operations.
- Simulate software crises.
- Operate a bug bounty program.
- Automate verification of operational infrastructure security.

Appendix B

BSIMM checklist at Ridder

In this appendix, we have documented which practices Ridder already does. The practices in bold are the ones they do.

B.1 Governance

B.1.1 Strategy and metrics

Level 1

- Publish process and evolve as necessary. Everyone knows the plan and is on the same page.
- Create evangelism role and perform internal marketing. Pick someone to help the teams build more secure software by talking with them.
- Educate executives to convince them to take security seriously.
- Identify gate locations, gather necessary artifacts. Determine what level of security should be required when releasing without enforcing it.

B.1.2 Compliance and policy

Level 1

- **Unify regulatory pressures. Combine all different regulations into one set of things to comply to.**
- **Identify Personally Identifiable Information obligations.**
- **Create policy to handle with regulations. Everyone should comply to the policy and don't have to keep checking all regulations.**

B.1.3 Training

Level 1

- Conduct awareness training.
- Deliver role-specific advanced curriculum.
- Deliver on-demand individual training.

B.2 Intelligence

B.2.1 Attack Models

Level 1

- **Create a data classification scheme and inventory to see which applications use the most data or most sensitive data. The higher classification uses more data and has a higher priority.**
- Identify potential attackers.
- Gather and use attack intelligence. Keep up to date with new attack models.

B.2.2 Security Features and Design

Level 1

- **Build and publish security features. Not every team needs to make the same features for authentication for example.**
- Engage the Software Security Group with architecture teams. Have someone from the SSG present at architecture meetings.

B.2.3 Standards and Requirements

Level 1

- Create security standards.
- Create a security portal that is easily accessible for everyone. Here you have all the information regarding security, possibly a wiki page maintained by the SSG.
- Translate compliance constraints to requirements.

B.3 SSDL Touchpoints

B.3.1 Architecture Analysis

Level 1

- Perform security feature review.
- Perform design review for high-risk applications.
- Have SSG lead design review efforts.
- Use a risk questionnaire to rank risks of applications.

B.3.2 Code Review

Level 1

- Have the SSG perform ad hoc review.
- **Use automated tools along with manual review.**
- **Make code review mandatory for all projects.**
- Use centralized reporting to close the knowledge loop and drive training. Bugs found during code review are stored in a central database so everyone can learn from them.

B.3.3 Security Testing

Level 1

- **Ensure QA supports edge/boundary value condition testing.**
- **Drive tests with security requirements and security features. For example checking that a user is timed out after repeatedly giving the wrong password.**

B.4 Deployment

B.4.1 Penetration Testing

Level 1

- Use external penetration testers to find problems.
- Feed results to the defect management and mitigation system.
- Use penetration testing tools internally.

B.4.2 Software Environment

Level 1

- Use application input monitoring to see what is given as input when the software is released to spot possible attacks.
- Ensure host and network security basics are in place.

B.4.3 Configuration Management and Vulnerability Management

Level 1

- Create or interface with incident response to react to incidents.
- Identify software defects found in operations monitoring and feed them back to development.

Appendix C

SonarQube Security Rules

In this appendix, we show the security-related rules that are activated in SonarQube at Ridder. SonarQube divides these rules in two categories: Vulnerability and Security Hotspot. Therefore, we kept the rules in the two separate tables below.

Vulnerablility (Title)	**Category**	**Language**
”CoSetProxyBlanket” and ”CoInitializeSecurity” should not be used	Vulnerabilityowasp-a6	C#
AES encryption algorithm should be used with secured mode	Vulnerability	C#
ASP.NET HTTP request validation feature should not be disabled	Vulnerability	C#
Cipher algorithms should be robust	Vulnerabilitycwe, owasp-a3, owasp-a6, privacy, sans-top25-porous	C#
Console logging should not be used	Vulnerabilityowasp-a3	C#
Cryptographic keys should be robust	Vulnerabilitycwe, owasp-a3, owasp-a9, privacy	C#
Database queries should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a1, sans-top25-insecure, sql	C#
Encryption algorithms should be used with secure mode and padding scheme	Vulnerabilitycwe, owasp-a3, owasp-a6, privacy, sans-top25-porous	C#
Endpoints should not be vulnerable to reflected cross-site scripting (XSS) attacks	Vulnerabilitycwe, owasp-a7, sans-top25-insecure	C#
Generic exceptions should not be ignored	Vulnerabilitycwe, error-handling, owasp-a10, suspicious	C#
HTTP request redirections should not be open to forging attacks	Vulnerabilitycwe, owasp-a5, sans-top25-risky	C#
HTTP response headers should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a7, sans-top25-insecure	C#
I/O function calls should not be vulnerable to path injection attacks	Vulnerabilitycwe, owasp-a1, owasp-a5, sans-top25-risky	C#
JWT should be signed and verified with strong cipher algorithms	Vulnerabilitycwe, owasp-a3, privacy	C#
LDAP connections should be authenticated	Vulnerabilitycwe, owasp-a2	C#
LDAP queries should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a1	C#
Logging should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a1, sans-top25-insecure	C#
Members should not have conflicting transparency annotations	Vulnerabilityowasp-a6, pitfall	C#
Mutable fields should not be ”public static”	Vulnerabilitycwe, unpredictable	C#
Neither DES (Data Encryption Standard) nor DESede (3DES) should be used	Vulnerability	C#
OS commands should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a1, sans-top25-insecure	C#
Regular expressions should not be vulnerable to Denial of Service attacks	Vulnerabilitycwe, denial-of-service, owasp-a1	C#
Serialization constructors should be secured	Vulnerabilityowasp-a8, serialization	C#
Server certificates should be verified during SSL/TLS connections	Vulnerabilitycwe, owasp-a3, owasp-a6, privacy, ssl	C#
Server-side requests should not be vulnerable to forging attacks	Vulnerabilitycwe, owasp-a5, sans-top25-risky	C#
SHA-1 and Message-Digest hash algorithms should not be used in secure contexts	Vulnerabilitycwe, owasp-a3, owasp-a6, sans-top25-porous, spring	C#
XML parsers should not be vulnerable to XXE attacks	Vulnerabilitycwe, owasp-a4	C#
XPath expressions should not be vulnerable to injection attacks	Vulnerabilitycwe, owasp-a1	C#

Security hotspot (title)	Category	Language
Changing or bypassing accessibility is security-sensitive	Security Hotspotowasp-a3	C#
Configuring loggers is security-sensitive	Security Hotspotcwe, owasp-a10, owasp-a3, sans-top25-porous	C#
Controlling permissions is security-sensitive	Security Hotspotowasp-a5, sans-top25-porous	C#
Creating cookies without the ”HttpOnly” flag is security-sensitive	Security Hotspotcwe, owasp-a7, privacy, sans-top25-insecure	C#
Creating cookies without the ”secure” flag is security-sensitive	Security Hotspotcwe, owasp-a3, privacy, sans-top25-porous	C#
Delivering code in production with debug features activated is security-sensitive	Security Hotspotcwe, error-handling, owasp-a3, user-experience	C#
Encrypting data is security-sensitive	Security Hotspotcwe, owasp-a3, owasp-a6, sans-top25-porous	C#
Expanding archive files is security-sensitive	Security Hotspotcwe, owasp-a5	C#
Formatting SQL queries is security-sensitive	Security Hotspotbad-practice, cwe, owasp-a1, sans-top25-insecure, sql	C#
Hard-coded credentials are security-sensitive	Security Hotspotcwe, owasp-a2, sans-top25-porous	C#
Hashing data is security-sensitive	Security Hotspotcwe, owasp-a3, owasp-a6, sans-top25-porous, spring	C#
Reading the Standard Input is security-sensitive	Security Hotspotcwe	C#
Using command line arguments is security-sensitive	Security Hotspotcwe, owasp-a1, sans-top25-insecure	C#
Using hardcoded IP addresses is security-sensitive	Security Hotspotowasp-a3	C#
Using pseudorandom number generators (PRNGs) is security-sensitive	Security Hotspotcwe, owasp-a3	C#
Using regular expressions is security-sensitive	Security Hotspotcwe, owasp-a1	C#
Using Sockets is security-sensitive	Security Hotspotcwe, owasp-a3, sans-top25-porous, sans-top25-risky	C#
Writing cookies is security-sensitive	Security Hotspotcwe, owasp-a3, sans-top25-porous	C#