Master's thesis

Master's Programme in Computer Science

Towards secure software development at Neste - a case study

Anton Moroz

September 30, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science

Tekijä — Författare — Author

Anton Moroz

Työn nimi — Arbetets titel — Title

Towards secure software development at Neste - a case study

Ohjaajat — Handledare — Supervisors

Dr. Antti-Pekka Tuovinen, Prof. Tomi Männistö

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year September 30, 2022	Sivumäärä — Sidoantal — Number of pages 47 pages

Tiivistelmä — Referat — Abstract

Software development industry has been revolutionized through adoption of software development methods such as DevOps. While adopting DevOps can speed up development through collaborative culture between development and operations teams, speed-driven adoption can have an adverse impact on security aspects. DevSecOps is a concept that focuses on embedding security culture and activities into DevOps. Another contributing factor to the more agile development landscape is the widespread adoption of open source components. However, the risk of putting too much trust into the open source ecosystem has resulted in a whole new set of security issues that have not yet been adequately addressed by the industry.

This thesis is commissioned by Neste Corporation. The company has set an initiative to incorporate methods that enable better transparency, agility, and security into their software development projects. This thesis collects research data on secure software development practices by combining findings of a literature review with a case study. The qualitative case study is done by interviewing eight stakeholders from four different software development teams.

The literature review shows that securing software is very much an ongoing effort, especially in the open source ecosystem. Therefore, it might be not surprising that the results from the case study revealed multiple shortcomings on the subject matter despite obvious efforts from the participating teams. As a result, this thesis presents potential ideas for the case company to consider integrating into their software development projects in order to kickstart their secure software development journey.

ACM Computing Classification System (CCS)
Security and privacy → Software and application security

Avainsanat — Nyckelord — Keywords

DevOps, DevSecOps, Software supply chain

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

DevOps, an acronym combining Development and Operations, is a set of principles and practices that emerged to break down silos between software development and IT operations teams by focusing on collaboration and shared responsibilities (Senapathi et al., 2018). The goal of DevOps is to support an agile software development life cycle through continuous software development practices (Senapathi et al., 2018). This means that one of the expected benefits, or drivers for teams adopting DevOps is achieving a competitive advantage by having the ability to respond quicker to customer needs through more frequent deployment of new features (Senapathi et al., 2018).

In the recent years, exploits by malicious actors of existing security vulnerabilities in software have highlighted the importance of integrating security activities in DevOps (Rajapakse et al., 2022). DevSecOps is a paradigm in which security efforts are integrated in DevOps by focusing on communication, collaboration, and integration between the development, operations, and security teams (Rajapakse et al., 2022).

A software supply chain can be defined as anything that is needed to deliver software, such as code, binaries, or a code repository (Kaczorowski, 2020). As the modern software development process relies heavily on potentially vulnerable open source software supply chains (SonaType, 2021), the growing interest for mitigating cybersecurity risks originating from software vulnerabilities is understandable. So much so that in May 2021, the US president signed an executive order in part to improve the US Nation's software supply chain security (Executive Office of the President, 2021).

This thesis is commissioned by Neste Corporation. The company has recently launched an initiative to better utilize existing cloud resources and incorporate better transparency, agility, and security into existing software development projects. The company relies almost exclusively on outsourcing in software development. Research done in this thesis focuses mainly on the security requirements for developers in the software supply chain. Therefore, the focal point of this thesis is to first document a select set of existing activities to practice secure modern day software development in the industry, then conduct a qualitative case study with a variety of members of Neste's outsourced software development teams, and finally create an actionable plan based on earlier observations which would act as the starting point in the journey to meet Neste's goals of the initiative.

The structure of this thesis is as follows. Section 2 introduces the research setting. Sections 3 and 4 motivate the case study by introducing continuous software development practices and industry recommendations for integrating security. Section 5 introduces the qualitative case study accompanied by interview results. The results to the research questions are presented and compared with existing literature in section 6. Section 7 concludes this thesis.

2 Research setting

This thesis is commissioned by Neste Corporation and this chapter introduces the reasons that motivate this study. Additionally, the research questions and methods are introduced to explain the structure of the following chapters of this thesis.

2.1 Background

Neste is mostly relying on outsourcing in custom software development, which supplements main business processes implemented in Enterprise Resource Planning (ERP) systems and other Commercial-off-the-shelf (COTS) systems. Unfortunately, operating with a relatively lean internal IT department has resulted in fragmented standards and practices between projects over time. This, in turn has introduced various impediments to reliable software delivery in Neste's projects. The company has recently launched an initiative to improve the situation, which can be illustrated as a roadmap, as shown in figure 2.1.

The roadmap is motivated by four key goals. The first goal is to ensure business continuity by taking control over the building blocks of code. This includes enforcing vendors to work under specific repository configurations, follow specific development guidelines and providing support on selected technologies. Additionally, use of Neste owned code repositories also helps to protect Neste's *Intellectual property rights* (IPR), which is the second key goal of the roadmap.

The third goal is to *set a certain quality of deliverables* by having vendors work under specific standards on topics mentioned in the roadmap, such as following security practices, and documenting the product from various viewpoints. This, in part, helps Neste *improve the speed and agility of deliverables*, which is the fourth goal of the roadmap.

The roadmap is divided into five phases that are built incrementally on top of one another: *Foundation, normalization, standardization, automation* & *expansion*, which eventually leads to automation-based *self-service.* The base provided by the foundation phase is done by other vendors included in this project, and therefore outside the scope of this thesis. Since the initiative is in its early stages, and the contents of the latter steps of the roadmap are still subject to change, the priority in this thesis is to discuss aspects of the

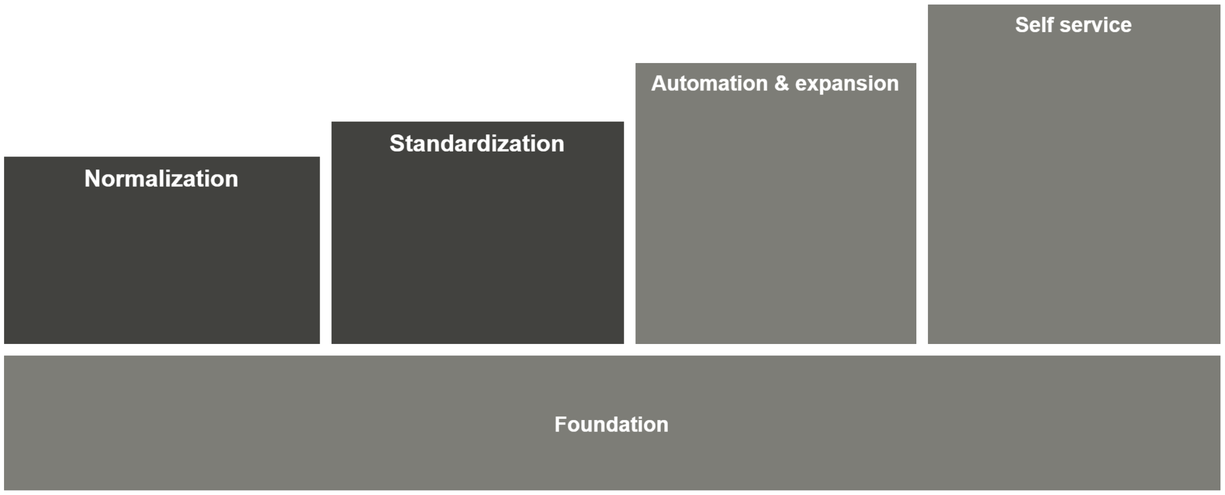

Figure 2.1: Neste's incremental roadmap for software development.

normalization and standardization phases of the roadmap.

The normalization phase is composed of activities that can be thought of as a set of building blocks to enable continuous and collaborative software development (e.g., version control, code reviews, project documentation). Team compositions change frequently in software consulting world compared to in-house development teams, which is why Neste would like to create new, and enhance existing processes and documents in order to onboard developers under the Neste way of developing software, as defined by the company. To properly implement these procedures, Neste would like to know more about vendors' currently implemented ways of working related to software development practices such as repository configurations, code review procedures, project architecture documentation, among others.

The standardization phase consists of software development technologies of which Neste would like have a better picture of in general. This includes things such as projects' current state of software testing, the CI/CD pipeline, and the technology stack. By understanding the current landscape of technologies used in existing projects, combined with learning more about the modern software development tools and practices, Neste would be able to invest resources appropriately and focus on the most relevant areas to provide technical guidance to help with both new, and previously existing software development endeavours.

2.2 Research questions

This thesis takes the dual approach of finding answers to research questions by utilizing both existing literature, and conducting qualitative research by interviewing various stakeholders involved in Neste's software development projects. With the previously presented background, it results in the following research questions:

RQ1: What are the commonly adopted practices for secure software development? In the recent years, there has been continuously increasing interest towards secure software. Finding an answer to this question consists of researching literature from topics such as DevOps, DevSecOps, application security, and software supply chains.

RQ2: How do Neste's software development teams currently address the state of code security in their projects? Answering this research question will be conducted through a qualitative case study which consists of developers, data engineers, product owners, and other stakeholders currently involved in developing software for Neste.

RQ3: Which code security measures should be integrated into existing Neste's software development projects? Security in the context of software development is a broad topic, therefore the focus will be on finding solutions to the most pressing security threats by combining the findings from researched literature with the results from the case study.

2.3 Research methods

2.3.1 Literature review

Security is a constantly evolving topic in the field of software engineering. The dynamic threat landscape results in continuously evolving ways to protect software. Therefore, to understand the current software security landscape, information in this thesis was collected from both academic literature and various types of grey literature. The primary academic literature included in this thesis was searched from Google Scholar, ACM Digital Library and IEEE Xplore digital library. The retrieved results were then selected based on search terms presented in table 2.1.

The structure for the literature review is divided into three parts: *DevOps*, *DevSecOps*, and *Software supply chains*. The primary peer-reviewed articles selected for each of these

Search string	**Range**	**Library**
DevOps challenges	2015-2022	ACM DL
DevSecOps challenges	2015-2022	Google Scholar
Software supply chain	2020-2022	Google Scholar

Table 2.1: Search keywords used to retrieve the main peer-reviewed articles.

topics is presented in table 2.2, in respective order. Insights within those academic articles are accompanied by other peer-reviewed and grey literature, where relevant. The search process for grey literature presented in this thesis was relatively unstructured. It includes a set of industry reports, insights from ongoing software development projects, and blogs of the current security landscape in software development.

Title	**Origin**
DevOps Capabilities, Practices, and Challenges: Insights from a Case Study	Senapathi et al., 2018
Challenges and solutions when adopting DevSecOps: A systematic review	Rajapakse et al., 2022
Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations	Enck and Williams, 2022

Table 2.2: Key articles retrieved from academic literature.

2.3.2 Case study

To investigate current events in their natural context, case study research is an appropriate research method (Runeson and Höst, 2009). The primary objective of the qualitative case study performed in this thesis was to understand the current state of software development practices and processes used by vendors developing software for Neste. From this thesis' perspective, some of the findings would be used to answer RQ2.

Data collection consisted mostly of conducting interviews directly with both technical (e.g., software developers), and non-technical (e.g., project managers) stakeholders involved in Neste's software development efforts. Teams with ongoing software development projects, which also had currently available stakeholders to interview were chosen to be included in this thesis. However, Robot process automation (RPA) and Integration development teams were excluded from this study. To achieve a more holistic picture, the goal was to interview at least two members from each team, but this was not done in every case. The time budget for each interview was 60 minutes. The interviews were recorded, and a written summary was saved and reviewed by the relevant interviewee to confirm findings and to correct possible misunderstandings. The interviews were semi-structured, and thus the answers varied between different interviews. As a result, we maintained contact with the interviewed teams to discuss some topics from new viewpoints as the result themes began to emerge.

3 Modern software development

To understand and address the currently existing security concerns in the software development landscape, first we need to understand the modern development workflow. This chapter introduces continuous software development practices by incrementally introducing terms such as DevOps, DevSecOps, and other ways to practice secure software development as recognized in the industry.

3.1 DevOps

The concept of DevOps has been previously challenging to define. However, it can be generally described as a set of various practices and principles that focus on bridging the gap between the software development and IT operations teams through collaborative means (Senapathi et al., 2018). The main purpose of DevOps is to support the agile software development lifecycle by employing continuous software development processes such as continuous delivery and microservices (Senapathi et al., 2018).

Adopting DevOps can be expensive and time-consuming. However, since the expected benefits, or drivers for implementing DevOps are greater than the costs of the implementation journey, many organisations justify the investment (Senapathi et al., 2018). To understand the realized benefits and challenges in DevOps, Senapathi et. al., conducted a qualitative case study. The case organization is a software company in the Finance/Insurance sector that was around one year into the DevOps adoption process at the time of the data collection. The data collection set consisted of in-depth interviews with key stakeholders that were responsible for DevOps implementation such as developers, testers, and relevant management stakeholders.

The identified drivers were grouped into three categories: Strategic, tactical, and operational, as displayed in Figure 3.1. The main strategic driver for adopting DevOps identified in the interviews was to achieve continuous deployment **(CD)** (Senapathi et al., 2018). CD is one of the continuous software development processes to accelerate the delivery of software without quality compromises (Shahin et al., 2017). CD also relates to other drivers directly such as being able to rapidly deploy bug fixes, or indirectly by being able to respond to customer needs quicker, as shown in Figure 3.1. One of the identified key

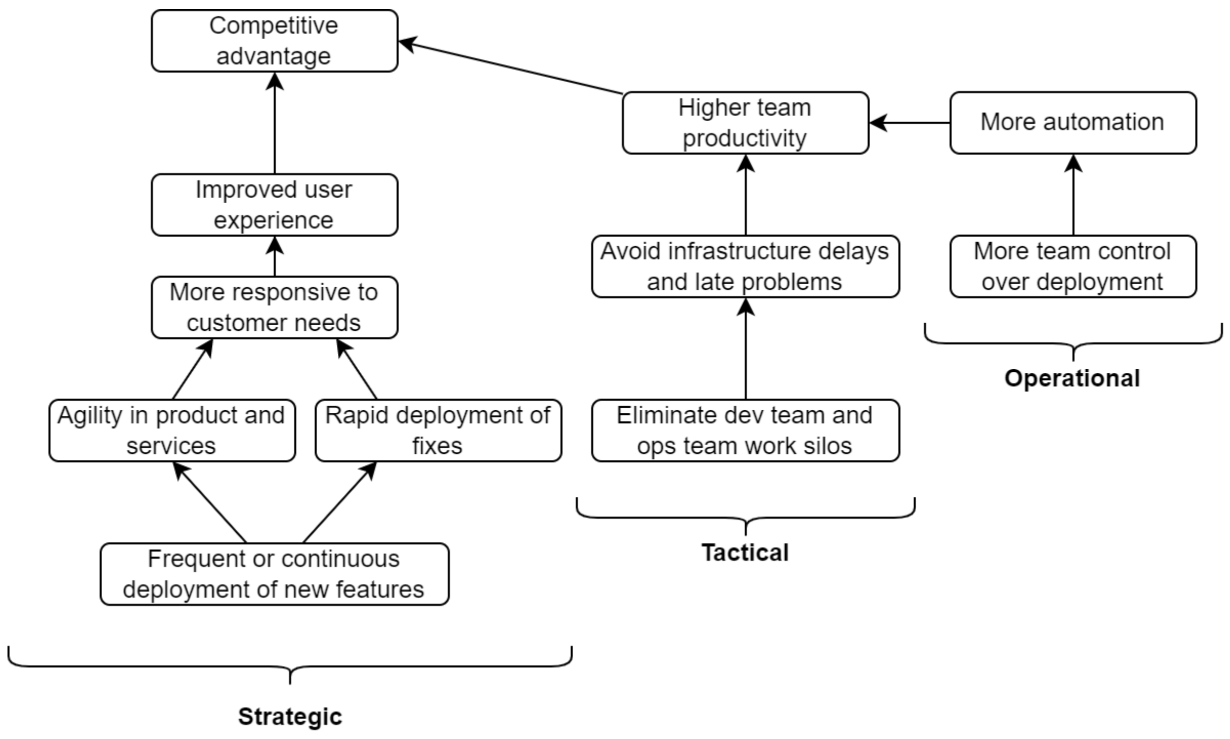
Competitive advantage
Higher team productivity
More automation
Improved user experience
More responsive to customer needs
Avoid infrastructure delays and late problems
More team control over deployment
Operational
Agility in product and services
Rapid deployment of fixes
Eliminate dev team and ops team work silos
Frequent or continuous deployment of new features
Tactical
Strategic

Figure 3.1: Figure of drivers for implementing DevOps. Adapted from (Senapathi et al., 2018).

tactical drivers was in productivity improvements through closer collaboration between the development and operations teams as the case organization had bottlenecks in getting features into production because of the existing silos between the teams (Senapathi et al., 2018). Finally, from the development team's point of view, a key operational driver was the ownership of the infrastructure instead of reliance on other teams to handle it (Senapathi et al., 2018). Together, these interconnected drivers would in theory, provide a competitive advantage, as shown in Figure 3.1.

Enablers can be defined as associated factors that support the DevOps way of working effectively (Senapathi et al., 2018). Enablers of DevOps can be divided into three categories: Capability enablers, technological enablers, and cultural enablers (Senapathi et al., 2018; Smeds et al., 2015). The capability enablers can be seen as the main DevOps enablers that can only work efficiently when supported by the technological and cultural enablers (Smeds et al., 2015). The individual enablers of each category are shown in Table 3.1.

Capability enablers include carrying out the activities of software development such as planning, development, testing, and deployment, in small increments and no delay, based on the feedback from the other activities (Senapathi et al., 2018; Smeds et al., 2015).

Capability enablers	Collaborative and continuous development
	Continuous integration and testing
	Continuous release and deployment
	Continuous infrastructure monitoring and optimization
	Continuous user behaviour monitoring and feedback
	Service failure recovery without delay
	Continuous measurement
Technological enablers	Build automation
	Test automation
	Deployment automation
	Monitoring automation
	Recovery automation
	Infrastructure automation
	Configuration management for code and infrastructure
	Metrics automation
Cultural enablers	Shared goals, definition of success, and incentives
	Shared ways of working, responsibility, and collective ownership
	Shared values, respect and trust
	Constant and effortless communication
	Continuous experimentation and learning

Table 3.1: Combined enablers of DevOps. Adapted from (Senapathi et al., 2018; Smeds et al., 2015)

Technological enablers support the capability enablers through task automation (Senapathi et al., 2018). Automation supports individual enablers such as continuous delivery by providing a single path to production for all changes to a system (Senapathi et al., 2018) and allowing employees to shift focus from the error-prone manual tasks to innovative and productive tasks (Smeds et al., 2015). Implementing technological enablers in an organization depends on tool choice, configuration, and design (Smeds et al., 2015).

Adoption of cultural enablers contributes to the capability enablers in a positive way as they emphasize collaboration, blameless work environment, awareness of common goals, and a climate for continuous experimentation and learning (Smeds et al., 2015; Senapathi et al., 2018). Unlike with technological enablers, adopting cultural enablers is not such a straightforward process, as time, effort, and resources are required for people to adjust to changes, and other improvement work (Smeds et al., 2015).

To achieve expected benefits of DevOps, implementing an automation pipeline to integrate new features is one of critical technological enablers (Senapathi et al., 2018). An automation pipeline, also known as CI/CD pipeline (RedHat, 2022), typically consists of two parts: *Continuous integration* combined with either *continuous delivery* or *continuous deployment*.

Continuous integration (**CI**) is a software development practice, where development team members integrate new development work frequently (Shahin et al., 2017; Fowler, 2006). CI enables software developers to achieve shorter and more frequent release cycles, produce higher quality software, and increase the productivity of the development teams (Shahin et al., 2017). More specifically, CI can be defined as a process in which an automatic trigger launches a series of interconnected steps such as compiling code, running tests, code coverage and coding standards validation, and building deployment packages to be handled in the following phases of the pipeline (Fitzgerald and Stol, 2014). An example representation of CI is shown in Figure 3.2, in which an event in form of a code update in source control management system, such as GitHub, automatically triggers a set of interconnected processes to prepare for the following phase in the CI/CD pipeline. In case the CI process fails, a number of protocols may take place to help solve the situation as quickly as possible (Fitzgerald and Stol, 2014).

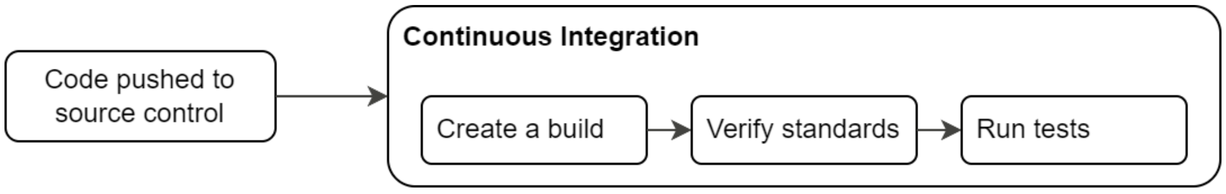

Figure 3.2: Continuous integration example.

CI by itself is not enough to move the changes to production, which is why it is followed by either continuous deployment (**CD**), or continuous delivery (**CDE**) (Fitzgerald and Stol, 2014). CDE performs additional tests (Shahin et al., 2018), and utilizes deployment automation to deliver software in a production-like staging environment. Its benefits include cost reductions, faster user feedback, and reduced deployment risk (Shahin et al., 2017). CD builds on top of CDE by automatically delivering software to production and customer environments (Shahin et al., 2017; Fitzgerald and Stol, 2014). The CDE practice can be applied to all types of systems, whereas CD may not be suitable for all cases (Shahin et al., 2017). For example, there might be regulatory requirements or business cases where a manual approval process is required before something can be released into production (Caum, 2013; Bird, 2015). An example representation of CDE/CD is shown in Figure 3.3.

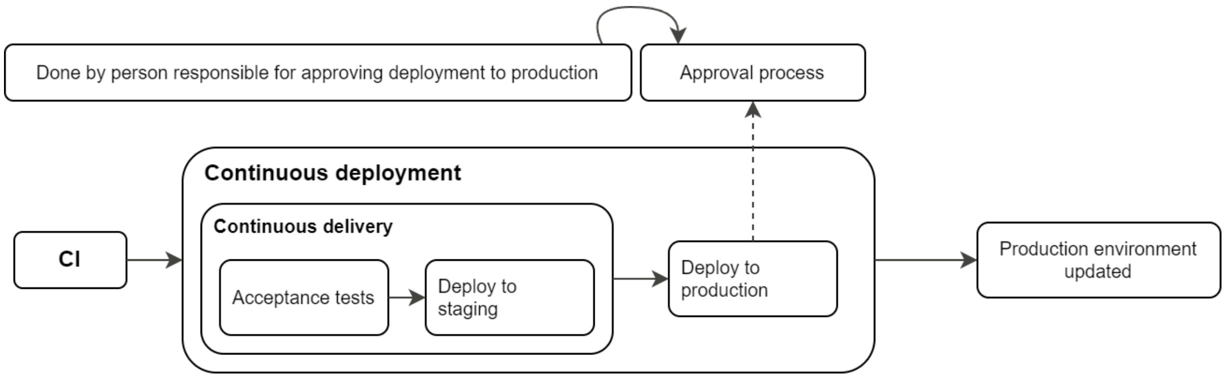

Figure 3.3: Example of continuous delivery with continuous deployment.

For the case organisation in the case study of Senapathi et. al., the main goal in implementing DevOps was to achieve continuous delivery in which the implementation of the CI/CD pipeline was a central aspect (Senapathi et al., 2018).

The set of actually realized benefits in DevOps can be similar to the expected ones. In the case organization, teams were happier and more engaged, and they were able achieve more frequent releases (Senapathi et al., 2018). However, adopting DevOps is not straightforward as previous literature has documented a variety of impediments that hinder adoption of DevOps such as challenges in organizational structure, buzzword tiredness, added amount of responsibilities, and knowledge requirements for both developer and operation stakeholders (Smeds et al., 2015). In the case study, some challenges were related to shortage of knowledgeable staff and resistance to change in the DevOps realm, while others were more closely related to cultural aspects (Senapathi et al., 2018). Organizations vary very much from one another from the perspective of the capability, technological, and cultural enablers. Therefore, it could be reasoned that the realized benefits and challenges from adopting DevOps could also be organization-specific.

3.2 DevSecOps

The benefit of being able to release software more frequently has been one of main factors for the wide adoption of DevOps in the software development industry (Rajapakse et al., 2022). As a result, one of the new challenges that has emerged is maintaining the agility benefits of DevOps while also considering the security aspects of the software. (Rajapakse et al., 2022). Previously, security activities have been handled at the later stages of the software development life cycle (**SDLC**). These activities, such as security code review or Dynamic Application Security Testing (**DAST**), are resource-intensive, which means that introducing these security activities in DevOps would come at the cost of speed when deploying software (Rajapakse et al., 2022).

DevSecOps can be defined as a principle that prioritizes security in the DevOps cycle (Rajapakse et al., 2022). Security measures are integrated by increasing collaboration between the development, security, and operations teams (Myrbakken and Colomo-Palacios, 2017; Rajapakse et al., 2022). Figure 3.4 illustrates the overview of the holistic integration of security in each phase of the DevOps cycle.

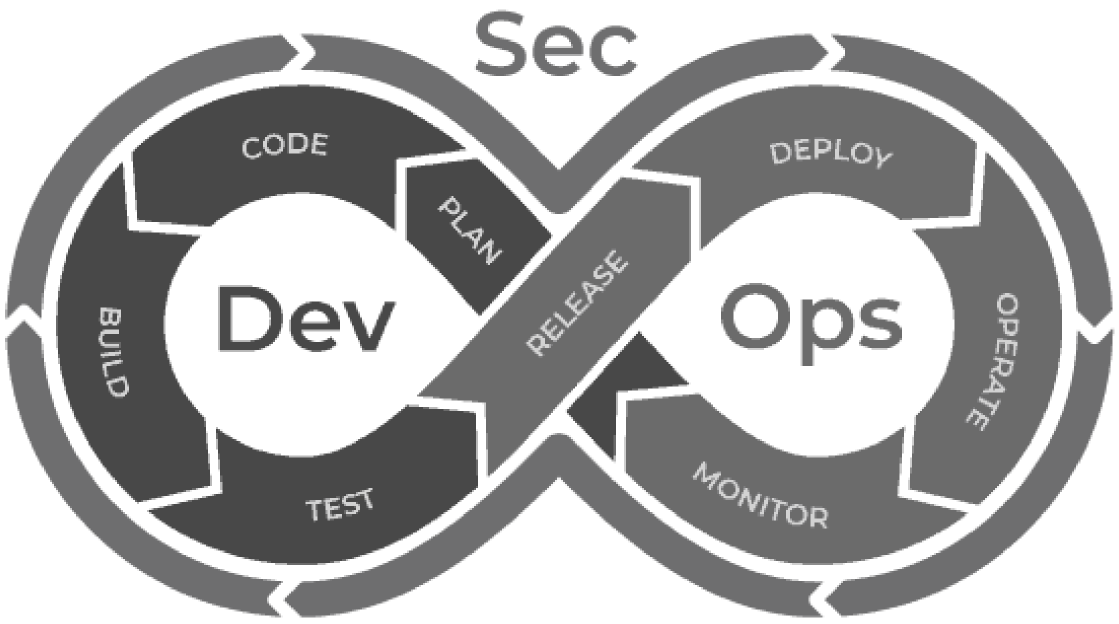

Figure 3.4: Illustration of DevSecOps. Adapted from (Plutora, 2019).

To better understand the challenges and solutions in DevSecOps, Rajapakse et. al., conducted a systematic literature review, in which the results were categorized into four interconnected themes: *People*, *practices*, *tools*, and *infrastructure* (Rajapakse et al., 2022).

Most challenges identified were tools-specific, and they can be divided into three different groups. First, the lack of standards in security tool selection, insufficient documentation of the already complex tools, and the configuration difficulty makes it challenging for developers to select or use the security tools in the first place (Rajapakse et al., 2022). Second, even though the established set of security tools bring certain benefits, developers prefer not to utilize them due their limitations and incompatibilities with the rapid deployment cycle of DevOps. The article mentions two types of tools related to this challenge group: Static Application Security Testing (**SAST**), and DAST tools. The code-based SAST tools detect software vulnerabilities by inspecting the source, or binary code without actually running the software (Rajapakse et al., 2022; Black et al., 2021). On the other hand, DAST tools are execution-based and require software to run in order to perform dynamic analysis (Rajapakse et al., 2022; Black et al., 2021). Furthermore, application security goes beyond only securing the source code. For example, the reliance on open source software in DevOps means that it is important to ensure that imported libraries are also secure. It is done through Software Composition Analysis (**SCA**) tools, that scan for vulnerabilities of third-party components (Rajapakse et al., 2021). One problem with SAST tools is the burdensome manual effort required to manage false positives. Another problem is the time it takes to scan code, along with high resource consumption, which is difficult to combine with the DevSecOps paradigm of incrementally adding small amounts of work (Rajapakse et al., 2022). DAST tools share rather similar difficulties with their counterparts in the sense that it takes time for the dynamic security tests to be completed, on top of also requiring a fair amount of manual effort to setup and run the tools in the first place (Rajapakse et al., 2022).

The third group of tools-specific challenges revolves around vulnerabilities arising from improper tooling configurations and usage in the DevSecOps pipeline. For example, while container technologies are popular in DevOps, the lack of security considerations, such as developers not paying attention to vulnerabilities in container images, or using insecure configurations and access control settings, have introduced security challenges (Rajapakse et al., 2022). In another example, the CD Pipeline itself is typically not designed with security requirements in mind, which makes the pipeline more vulnerable to attackers (Rajapakse et al., 2022).

A key recommendation to combat tools-specific challenges presented in the article is to utilize tooling specifically catered to DevSecOps, preferably with a cloud-based solution (Rajapakse et al., 2022). Selecting the appropriate tools is an ongoing challenge in the industry as a global research survey conducted by Enterprise Strategy Group (ESG) indicates that instead of dealing with the shortcomings of traditional security tools and tool sprawl, organizations are interested in consolidated solutions for web application and API security, that also integrate with the DevOps tools and processes, while also managing to be up-to date of the dynamic threat landscape (Grady and DeMattia, 2021).

Rajapakse et. al., identified numerous practices-specific challenges, which could be further divided into two groups. First, the difficulty to automate traditionally manual security practices, such as compliance testing, into the automation-oriented DevSecOps workflow is one of the critical challenges identified in the paper (Rajapakse et al., 2022). Second, developers have to balance the trade-offs between speed and security in DevOps. For example, the lack of appropriate tools and methods, combined with the fast delivery cycle in continuous deployment makes it difficult to rigorously verify security requirements (Rajapakse et al., 2022). Additionally, continuous security assessment was identified as one of the key practices for DevSecOps. However, processes associated with the practice have not been widely adopted in the industry. This is partly due to lack of instructions and lack of consensus on how security measures should be included in the pipeline (Rajapakse et al., 2022). These types of challenges, combined with the increased development speed, partly due to increased use of open-source components, cause security assurance to be more difficult. This in turn, results in some organizations perceiving DevOps and security aspects incompatible with each other, which then adds to their reluctance to adopt DevOps processes altogether (Rajapakse et al., 2022).

One of the ways to tackle practice-related challenges is to include security from the beginning of the development process. This concept is known as *shifting security left*, which is also one of the key recommendations in DevSecOps (Rajapakse et al., 2022). From the development standpoint, one of the ways it to integrate security scanning into the CI/CD pipeline to enable development teams to find and remediate possible vulnerabilities in the earlier stages of the SDLC (Wegner, 2020). An example is shown in figure 3.5. To succeed in shifting security to the left, developers and security teams must co-ordinate in all stages of the SDLC, as just adding new tools to the pipeline adds to the list of responsibilities to the already resource-constrained development teams that deal with their own challenges, which then leads to unused tooling instead of empowering development teams (Bell, 2022).

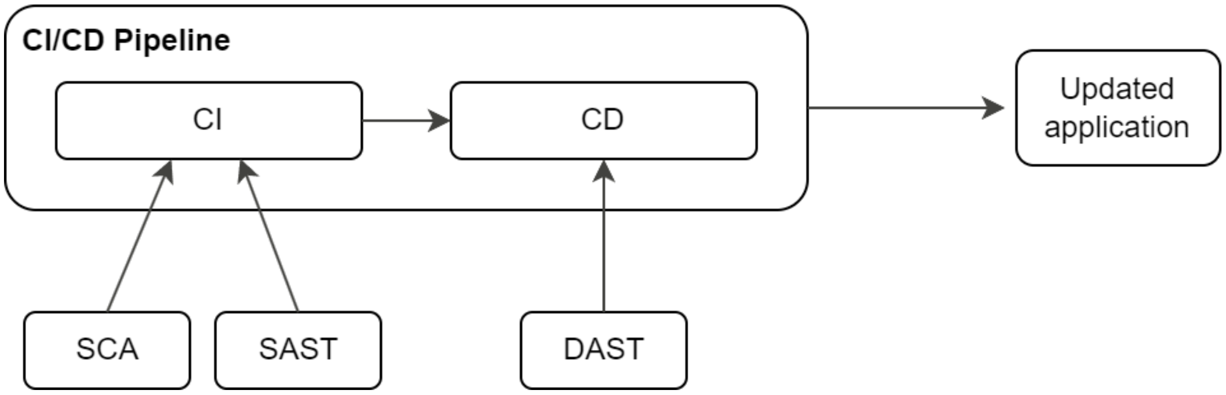

Figure 3.5: An example of possible tools used to shift security left in the CI/CD pipeline, which eventually triggers an event that updates an application.

To secure source code, shifting security to the left is more than just scanning code against vulnerabilities. A whitepaper by Google Cloud suggests a set of activities to apply for securing source code in the SDLC, and thus shift security to the left from another perspective (Ensor and Stevens, 2021). These activities are summarized in Table 3.2 below.

Automated testing	Automated tests lower security risks indirectly by providing a way to respond quicker to threats, discovered vulnerabilities, and regression defects.
Memory-safe languages	A majority of vulnerabilities patched by a security update are associated with memory safety issues. Therefore, teams should prefer memory-safe languages to reduce risk of memory-based vulnerabilities. If conditions do not permit using a memory-safe language, fuzz testing is recommended to identify vulnerabilities.
Flawless change management	A best practice is to perform code reviews that integrate the work of short-lived feature branches to the protected primary branch. To identify the appropriate set of code reviewers, some source code management (SCM) systems provide a CODEOWNERS file to identify parties responsible for different sections of the codebase.

Commit authenticity	Verify code contributors' authenticity by enforcing contributors to digitally sign their commits.
Identifying malicious code	Use static code analysis, or linting to identify and remediate vulnerabilities originating from common syntax mistakes, such as unused variables, array index overruns, and improper object references. To look for code functionality and logic errors, use automated testing tools that provide feedback through the CI/CD pipeline.
Avoid exposing sensitive information	Use pre-commit hooks to identify the potential exposure of sensitive information, such as passwords and API keys.
Logging and build output	To mitigate risks of leaking sensitive logging output in the CI/CD pipeline, hide embedded secrets through built-in tooling of CI, or by building scripts independently from CI.
License management	Use license scanning tools to prevent financial and legal ramifications due to license restrictions of open source software dependencies.

Table 3.2: Summary of securing source code (Ensor and Stevens, 2021).

As previously mentioned, one of the other key recommendations for DevSecOps is to implement continuous security assessment (Rajapakse et al., 2022). Similar to shifting security left, it involves continuous co-ordination between development and security teams, but also extends to activities after deploying the software. One such practice is continuous monitoring (**CM**) (Rajapakse et al., 2022). Continuous monitoring is especially useful in highly regulated environments, where tracing multiple types of input is important (Rajapakse et al., 2022). CM can be implemented by utilizing Runtime application self-protection (**RASP**), and Web application firewalls (**WAF**) (Rajapakse et al., 2021). RASP tools can detect attacks in real-time by continuously monitoring the run-time environment, while a WAF is deployed in production to monitor and take action against external attacks (Rajapakse et al., 2021). These techniques are complementary to each other. For example, WAFs can mitigate a Distributed denial-of-service (DDoS) attack by inspecting incoming web traffic, while RASPs have visibility into the application layer, and thus can provide protection within the application (Pasha, 2021).

Finally, the situation of developers dealing with the potentially vast amount of alerts from the decoupled security tools in the CI/CD pipeline can result in *alert fatigue* (Rajapakse et al., 2021). Therefore, it is recommended to use a Security information and event information (**SIEM**) platform to deliver a holistic picture of all security events in the CI/CD pipeline that would provide the developers sufficient information to remediate prevalent issues (Rajapakse et al., 2021).

Next, the amount of identified infrastructure-related challenges in the article by Rajapakse et. al., were smaller in numbers than each of the previously mentioned themes of tools and practices, but were still important. Adopting DevSecOps in a highly regulated (e.g., air-gapped), resource-constrained (e.g., IoT, embedded systems), or otherwise complex infrastructural setting is challenging due to various restrictions that conflict with the DevSecOps way of working (Rajapakse et al., 2022). Due to the nature of software products that Neste delivers to the oil refineries, insights to manage infrastructural challenges in highly regulated environments are very relevant from the development perspective. Rajapakse et. al., propose a few solutions to address this need. First, teams should consider adopting strict access management policies to allow each of the development, operations, and security team members only the most necessary access to sensitive environments (Rajapakse et al., 2022). Second, adopting Infrastructure as code (**IaC**) is also recommended, as utilizing IaC brings certain benefits. For example, IaC allows for pre-configured settings to be applied in systems in a centralized, and repeatable manner. Furthermore, it enables infrastructure itself to be managed similarly to what is done in software development in general (Rajapakse et al., 2022). Third, as software today is composed of a variety of components, including open source software (**OSS**), component-specific vulnerabilities should be systematically managed in a transparent process (Rajapakse et al., 2022).

The fourth and final theme of challenges identified by Rajapakse et. al., was centered around people. One important challenge is that developers who act as the centerpiece of handling software security, lack the necessary knowledge to do so (Rajapakse et al., 2022; Rajapakse et al., 2021). One reason for this is the lack of sufficient security education and training in software engineering (Rajapakse et al., 2022). The other challenge revolves around the inability to adopt the cultural changes required for DevSecOps (Rajapakse et al., 2022). For example, in some organizations, security is not necessarily seen as something that brings value, which adds to the reluctance to prioritize security measures (Rajapakse et al., 2022). However, the most prominent issues were related to inter-team collaboration, which exist due to frictions between the development and security teams, and the developers' siloed work culture preventing the collaborative way of working (Rajapakse et al., 2022).

To combat challenges related to people in DevSecOps, organizations should focus on enabling collaboration and cross-functionality. One key proposal is to introduce security champions (Rajapakse et al., 2022). Security champions should be dedicated and technically apt stakeholders, such as developers, that contribute to both software development and security activities in the SDLC (Jaatun and Soares Cruzes, 2021). The importance of selecting a competent member should not be neglected. For example, it is easier to train a developer to address security issues instead of teaching software to a security expert (Jaatun and Soares Cruzes, 2021). Other general suggestions related to people-centric challenges include training developers, sharing security knowledge, and utilizing communication channels that enhance inter-team collaboration (Rajapakse et al., 2022). For example, instead of relying on communication by individual emails, teams should leverage automation by having relevant stakeholders receive notifications automatically from activities of various processes, such as successful installations (Rajapakse et al., 2022).

Finally, it should be re-emphasized, that the previously mentioned four themes of DevSecOps challenges and solutions are interconnected. Adopting DevSecOps is not straightforward, and requires a significant culture change within an organization in order for practices and tools to be effective. Developers should be provided necessary guidance and developer-first tools to start shifting security to the left, and enable continuous security assessment, which are the two key recommendations in DevSecOps (Rajapakse et al., 2022).

4 Open source software and software supply chains

Open source software has revolutionized the way software is developed today. However, it's popularity introduces additional considerations, which requires us to pay attention to security beyond the CI/CD pipeline. This chapter focuses on the risks of careless use of open source software through the introduction of software supply chains and why securing them has become such an important topic in the industry today.

4.1 Software supply chains

A software supply chain consists of anything that goes into, or affects software, such as code, repositories, or package managers (Kaczorowski, 2020). The current way of developing software is heavily reliant on the open source community, and thus dependent on software supply chains managed by third parties (Kaczorowski, 2020).

According to a recent industry report by Synopsys, 97% of commercial codebases contain open source code (Synopsys, 2022). Moreover, the surveyed codebases themselves were largely comprised of open source (Synopsys, 2022). High dependency of application functionalities being provided by third parties results in increased security risks and exposure to vulnerabilities (Kaczorowski, 2020). To understand and manage the security risks of a software supply chain, first we need to understand the anatomy of one in more detail. It can be represented as a chain of interconnected steps that transform the developers' original artifact into one to be used by the consumers, as shown in figure 4.1.

A software supply chain consists of *artifacts* and *processes* that run on *platforms*. An artifact is a blob of immutable data, such as a git commit or a container image (SLSA, 2022b). A source refers to an artifact (e.g., a git commit) that was authored, or reviewed without further modifications, which is hosted on a platform (e.g., GitHub, Azure DevOps). It is the starting point of a software supply chain (SLSA, 2022b). The first step is followed by a set of input artifacts going through the build phase of the CI/CD pipeline, which results in a set of output artifacts (SLSA, 2022b). The input artifacts may originate from

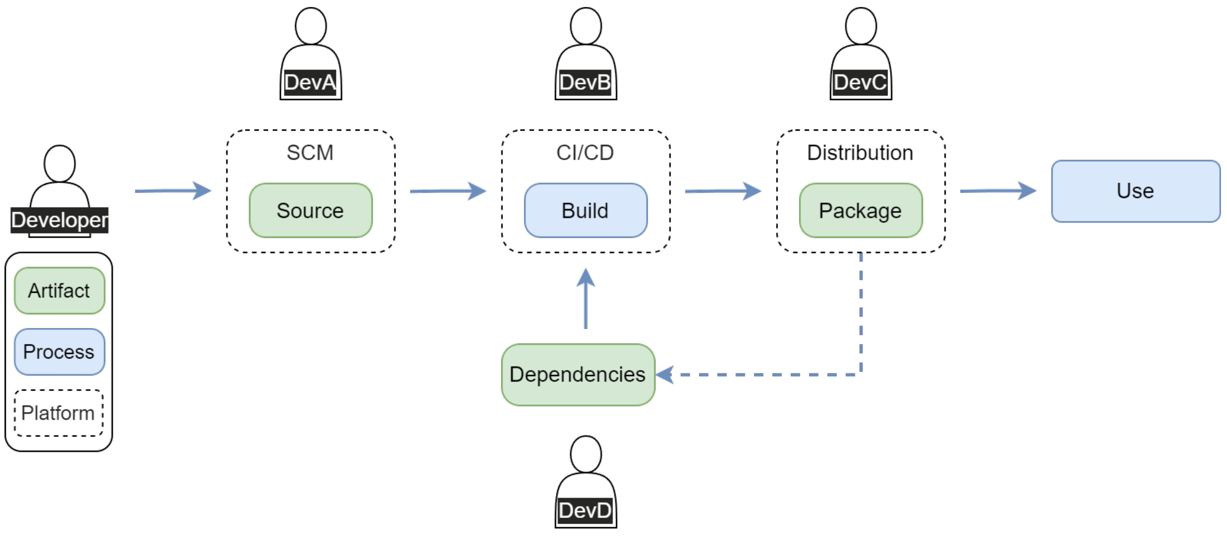

Figure 4.1: A visual representation of a software supply chain. Adapted from (SLSA, 2022b).

the source or dependencies. Finally, a package refers to an output artifact of the build process (e.g., Docker image), which is distributed for consumer use through a platform (e.g., DockerHub, Google container registry) (SLSA, 2022b).

A software supply chain can be attacked from any point, as shown in figure 4.2. For example, a developer can intentionally submit bad code. In one instance of this, as a part of their research, a group of researchers from the University of Minnesota submitted bad code into the Linux kernel by stealthily introducing vulnerabilities disguised as minor code patches, also known as *hypocrite commits* (Holz and Oprea, 2021; Lewandowski and Lodato, 2021; Cook, 2021). This experiment gathered negative attention and eventually resulted in the university itself getting banned from contributing to the Linux kernel altogether (Cook, 2021).

Software supply chain attacks can also happen without a directly malicious intent. In one case, a bad version of another package was integrated into the dependencies of the event-stream package in the Javascript ecosystem by a malicious actor, who was supposed to be the new person to handle the maintenance of event-stream. This situation resulted in new installations of event-stream to be infected (Kaczorowski, 2020; Grander and Tal, 2018). In this case, the infected dependency used by the event-stream package would steal Bitcoin and Bitcoin cash cryptocurrencies from users under a set of very specific circumstances (Arvanitis et al., 2022).

The impacts of the previously mentioned software supply chain attacks were relatively limited. However, some attacks can have massive consequences. In the case of the So-

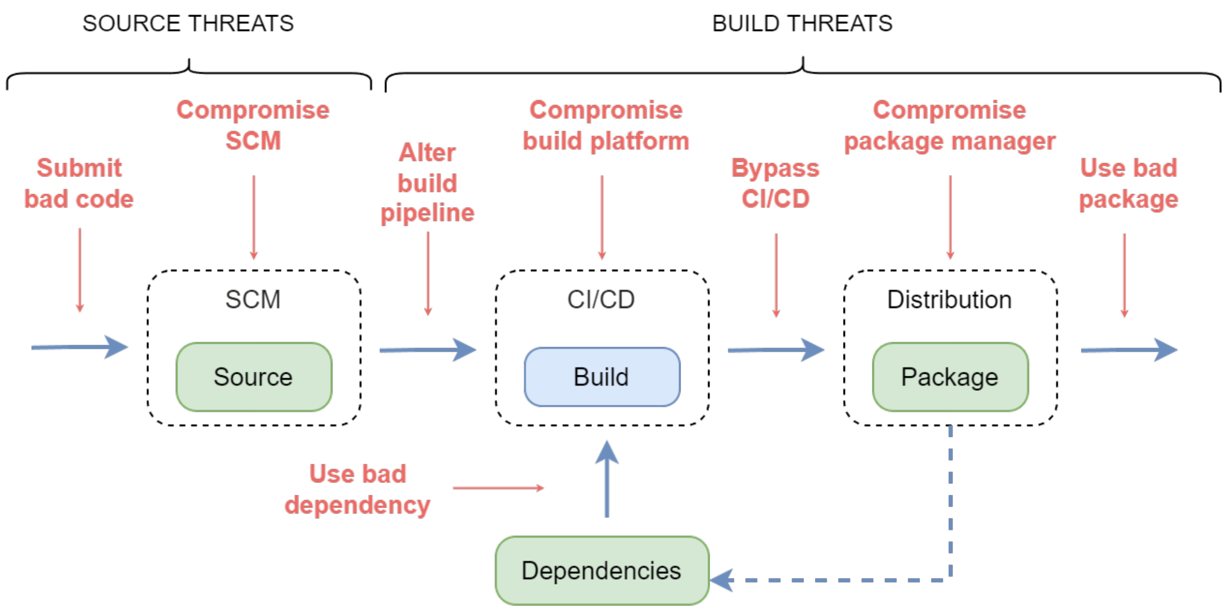

Figure 4.2: Software supply chain, including threats. Adapted from (SLSA, 2022c).

larWinds attack, the compromisation of SolarWinds' build platform resulted in malware being enabled to spy on over 100 companies and multiple U.S government agencies (Enck and Williams, 2022). Another software supply chain attack that has recently had an industry-wide effect was the log4j attack, in which the compromised logging library in the Java ecosystem allowed for remote code execution (Enck and Williams, 2022).

4.2 Securing software supply chains

The increased amount and severity of software supply chain attacks has prompted the U.S Government to issue an executive order to improve the nation's cybersecurity (Executive Office of the President, 2021; Enck and Williams, 2022). To understand the most pressing issues in the software supply chain security, Enck and Williams held three summits to collect practitioners' experiences and insights regarding the topic (Enck and Williams, 2022). As a result, they present a list of five top challenges.

The first challenge is *updating vulnerable dependencies* (Enck and Williams, 2022). There is debate between developers and security experts on whether to use fixed dependencies or not. The argument from developers' side is that with fixed dependencies, project-breaking changes are prevented (Enck and Williams, 2022). However, security experts push for automatic dependency updates as they enable being up-to date with possible security fixes (Enck and Williams, 2022). This is supported by the estimation that more than 85% of disclosed open source vulnerabilities have an already available solution (Kaczorowski, 2020; WhiteSource, 2020). However, the SolarWinds attack showed that dependency updates can also be malicious. In an ideal situation, updating dependencies would happen after it is confirmed that the update is safe, but also not too late, as the update might also remediate vulnerabilities (Enck and Williams, 2022). Another suggestion is to take a proactive approach by focusing on isolation techniques that limit the impact when a vulnerable dependency is exploited (Enck and Williams, 2022). One example of this is the RLBox framework (Enck and Williams, 2022), which mitigates the impact of compromised browser dependencies by sandboxing third-party libraries in the browser renderer (Narayan et al., 2020).

The second challenge is *leveraging the software bill of materials for security.* Similar to a list of food ingredients, software bill of materials (**SBOM**) is a formal record that contains details and supply chain relationships of components that were used to build software (Executive Office of the President, 2021). In theory, SBOMs can provide a variety of benefits by providing transparency to different stakeholders (Enck and Williams, 2022). For example, a company using an open source component in their application with an especially restrictive license can potentially result in the company being legally forced to make their application code public (Synopsys editorial team, 2016). SBOMs can help with preventing this issue by providing license information of a software component for it to evaluated before use. An example SBOM excerpt is shown in Listing 1.

```json
{
  "bomFormat": "CycloneDX",
  "specVersion": "1.4",
  "serialNumber": "urn:uuid:<EXAMPLE-UUID>",
  "version": 1,
  "components": [
    {
      "type": "library",
      "group": "com.example",
      "name": "test-utils",
      "version": "1.2.3",
      "licenses": [
        {
          "license": {
            "id": "Apache-2.0",
            "text": {
              "contentType": "text/plain",
              "encoding": "base64",
              "content": "<EXAMPLE-CONTENT>"
            },
            "url": "https://www.apache.org/licenses/LICENSE-2.0.txt"
          }
        }
      ]
    }
  ]
}
```

Listing 1: SBOM metadata example using the CycloneDX standard.

As a whole, from the builders' (e.g. software developers) perspective, an SBOM can help ensure that third-party components are up-to date, or improve responsiveness to new vulnerabilities (Executive Office of the President, 2021). Additionally, builders can reduce attack surface of the application by removing redundant components (Carmody et al., 2021). Second, software buyers can utilize SBOMs to evaluate the risk of builders' product, for example through a vulnerability analysis (Executive Office of the President, 2021; Carmody et al., 2021), and thus help making a better decision to select the best option for themselves. Third, SBOMs can enable people who operate and maintain the software, to proactively address newly discovered vulnerabilities, and determine whether their product or organization is at risk (Executive Office of the President, 2021; Carmody et al., 2021).

In reality, the current state of SBOMs is that their potential is largely unrealized. Practitioners view SBOMs as a list of ingredients and a compliance exercise (Enck and Williams, 2022). However, there are ongoing efforts, such as the CycloneDX project by the Open Web Application Security Project (OWASP) community, to enhance viability of SBOMs in the industry (OWASP CycloneDX, 2022). Either way, providing an SBOM is a compliance requirement for vendors wanting to sell software to the U.S government (Enck and Williams, 2022).

The third challenge is *choosing trusted supply chain dependencies.* The software supply chain is an exercise in trusting people who built the external dependencies (Enck and Williams, 2022). Currently there is a list of issues that need to be addressed for establishing trust with developers building the dependencies (Enck and Williams, 2022). These concerns are related to topics, such as trusting the dependency maintainers over a longer time period, or trusting them not hand the dependency itself over to a malicious actor (Enck and Williams, 2022).

These, and other similar concerns have resulted in relevant industry stakeholders such as package managers, researchers, and projects such as the Open Source Security Foundation (OpenSSF) working on mechanisms and products, which focus on separating the wheat from the chaff (Enck and Williams, 2022). For example, some tools aim to identify *typosquatting* (Enck and Williams, 2022), a technique where a malicious package is presented as the original. One instance of this is attackers releasing the malicious package under the same name as the target package in an alternative package repository (Ohm et al., 2020).

The fourth identified challenge is *securing the build process.* CI/CD tools, such as Jenkins and Github Actions, have become popular tools to run the build process (Enck and

Williams, 2022). However, security aspects related to these tools have been largely overlooked (Enck and Williams, 2022). For example, the CI/CD tasks provided by the open source community are not always designed with security in mind. This increases the attack surface by potentially allowing malicious code to be injected in the build process (Enck and Williams, 2022).

Supply chain Levels for Software Artifacts (SLSA) is a framework to ensure the integrity of the software supply chain (Enck and Williams, 2022; SLSA, 2022a). As of July 2022, SLSA is in the alpha stage, and thus under active development. The framework consists of security guidelines to offer protection against common software supply chain attacks (Lewandowski and Lodato, 2021). These guidelines focus primarily on two categories: *source*, and *build* integrity, as visualized in figure 4.2. One example related to improving build integrity is having stronger security controls for the build platform, which would have helped mitigate risks present in the SolarWinds attack (Lewandowski and Lodato, 2021). Another guideline related to the source integrity recommends implementing a thorough code review process. This, in turn, would have helped mitigate risks with the hypocrite commits presented earlier in this chapter (Lewandowski and Lodato, 2021). As a whole, SLSA is divided into four incrementally evolving levels. Achieving higher levels increases confidence that the software artifact has not been tampered with (Lewandowski and Lodato, 2021). A short summary of these levels is provided in table 4.1.

Level	**Description**	**Example**
1	Documentation of the build process.	Unsigned provenance.
2	Tamper resistance of the build service.	Hosted source/build. Signed provenance.
3	Extra resistance to specific threats.	Security controls on host. Non-falsifiable provenance.
4	Highest levels of confidence and trust.	Two-party review. Hermetic builds.

Table 4.1: Summary of SLSA levels (SLSA, 2022d).

By reaching the requirements of the first SLSA level through an automated and provenance generating build process, software consumers can begin making risk-based security decisions (Lewandowski and Lodato, 2021). Provenance in SLSA refers to metadata containing the origins of consumed software (Palafox, 2022), and it should be published alongside the software artifact it relates to (Blit, 2022).

The fifth, and final identified challenge is *getting industry-wide participation*. Large tech organizations are well-aware of the risks in the software supply chain. Some short-term risk management solutions exist, such as providing developers with repositories with pre-approved dependencies to choose from (Enck and Williams, 2022). However, short-term solutions are not viable in the long run. Therefore, industry stakeholders such as the Linux Foundation, are currently involved with projects such as the Open Source Security Foundation (OpenSSF) to enhance the security of the software supply chain (Enck and Williams, 2022). However, merely creating these tools and methods is not enough as they need to be adopted by the broader software industry, which is an ongoing challenge (Enck and Williams, 2022).

5 Case study

This chapter introduces the qualitative case study by presenting background of each participant and giving a summary of the interview results. The results are then categorized to fit the structure of a software supply chain.

5.1 Participants

TeamOne

TeamOne is in charge of a client-server project that contains stakeholders from different vendors (e.g., a software developer from one and a project manager from another). The project itself has existed for multiple years, but the team composition has changed relatively recently, as the previous development team has been succeeded by a completely new team. The current team consists of two client-side developers and five server-side developers. Developers are accompanied by a dedicated tester, and two UI/UX designers. The team is split into two responsibility groups; new features and maintenance. However, these two groups are cross-functional, and thus not strictly siloed.

TeamTwo

TeamTwo is part of Neste's subsidiary company that is responsible for ongoing development connected to Neste's production facilities. The software development team has a Research and development (R&D) programme, which enables the team to develop customer-specific projects. The product portfolio includes products and projects, such as client-server applications and operator training simulators. Due to security and real-time requirements, their software runs mostly in environments without internet connectivity, and on-premises in general. This means that most of their client-server applications are different compared to web and mobile applications today, in addition to little utilization of commercial cloud offering as a whole.

TeamThree

TeamThree is a small team consisting of two members. They are primarily responsible for the maintenance of a set of Neste's data analytics services that run on a public cloud platform. Additionally, a few web applications have also been merged into their scope of responsibilities.

TeamFour

TeamFour is responsible for development of Neste's common data platform. Their overall lineup consists of data engineers and a team providing the data required for data engineers to parse through. These two groups meet together in daily standups. Additionally, the team collaborates with many other stakeholders when needed, such as external api data providers.

Id	**Team**	**Role**	**Format**
T1-SW-1	TeamOne	Software development	A
T1-PM-1	TeamOne	Project management	A
T2-SW-1	TeamTwo	Software development	B
T2-AD-1	TeamTwo	System administration	B
T3-DE-1	TeamThree	Data engineering	A
T4-DE-1	TeamFour	Data engineering	A
T4-DE-2	TeamFour	Data engineering	A
T4-DE-3	TeamFour	Data engineering	A

Table 5.1: Participating individuals in this case study.

5.2 Results

5.2.1 Overview

At first glance, the interviewed set of teams follow a number of software development practices that are commonly associated with modern software development. For example, version control is used where possible, code review procedures exist, and teams are mostly using CI/CD pipelines to push code to production. However, the interview data revealed that there was quite a bit of variance between answers in regards to the extent these practices were used. This is understandable due to the differences between teams and their developers' backgrounds.

The interview formats resulted in more data being retrieved from the interviews than what is presented within this chapter. There are a few reasons behind excluding some information. One is that the conducted interviews needed to serve two purposes. First, they needed to provide Neste with information on development teams' workflows and technology stacks in general, while also providing enough data to meet the requirements on answering the research question RQ2. Additionally, the interviews began before the final direction of this thesis was possible to be selected. A different interview format was chosen for the interviewees working for Neste's subsidiary company because the subsidiary has a different role with Neste compared to outsourced vendors.

The result categories used to provide an answer for RQ2 were *Code reviews*, *Code security in the CI/CD pipeline*, and *Open source code dependency management*. These choices were motivated by the structure of a software supply chain, as presented in figure 4.1, combined with the presence of the CI/CD pipeline throughout the literature study of this thesis.

5.2.2 Main findings

Code reviews

Code reviews were a widely used practice according to the answers received from the interviewed participants. They were typically used as a code quality gate before pushing new work to production. A common scenario to trigger a new code review was by creating a pull request. As a whole, the typical code review was a manual process among the interviewed teams, and there was no unified standard on how to conduct one. Branching strategies between different projects were generally set up in a way that discouraged pushing new work directly to production, or otherwise skip code reviews. However, not every teams' code repositories had been explicitly configured to enforce a specific way of working.

It is important to note that the meaning of code varied between projects. Software development projects consisted of custom code through the usage of established programming languages (e.g., Java, Javascript, Python), whereas data engineering projects were also making significant use of proprietary tools and technologies. This meant that the code artifacts in the data engineering projects had differences when compared with the more traditional software development projects. As a result, software developers could be paying attention to different things compared to data engineers when conducting a code review. For example, one software developer gave the following answer when asked on what they were generally looking for when assigned to do a code review:

"...I try to focus on whether there are any obvious mistakes..."

On the other hand, validating that the new changes were working as intended in a data engineering project was not necessarily possible by just looking at the code artifact in all situations. Participants from two separate data engineering teams reported that they were conducting code reviews by manually testing parts of their work in a pre-production environment instead of analyzing code. This could be explained by the lack of suitable code analysis tools for code artifacts linked to proprietary technologies.

One interviewee expressed concerns regarding the challenge of project information being hidden from relevant stakeholders within their team. This is due to the team's multi-vendor team structure, where managers and developers work for Neste from different companies. One apparent problem was management's lack of visibility in developers' code reviews and similar. The team's project manager wished for better integration between their general communication channel with the used code repository, and commented the situation as follows:

"...feels kind of weird that there are systems to which Neste has no access, and where things related to their projects are being managed..."

In summary, while the code review processes lacked common structure, it was evident that the development teams generally strived towards shared responsibility and collective ownership of their code, which are some of the enablers in DevOps, as mentioned earlier in this thesis.

Code security in the CI/CD pipeline

Usage of application security tools in the CI/CD pipeline was in complete contrast compared to using code reviews to aid in achieving a certain level of code quality. None of the interviewed participants reported to use tooling meant to specifically scan for vulnerabilities. There were multiple reasons behind this. One reason is that the scanning tools do not always fit the technology stack. An example of this is a data engineering project, where the source code consisted mostly of code artifacts related to databases (e.g., SQL files ending with .db), and proprietary technologies (e.g., files ending with .json). One interviewee commented on this scenario as follows:

"...I'm not sure if they (application security tools) are viable for the *<anonymized data engineering product>* setup itself, maybe they are viable for SQL artifacts, the R artifacts, the Python artifacts..."

Additionally, one data engineer remarked that the differences between the backgrounds of data engineers and software developers could also play a role with development capabilities:

"...The security side is less at the forefront. The focus is on the data side. Quite a few of our developers are data scientists, so their background is completely different..."

On the other hand, software development projects faced their own set of impediments in adopting application security tools. One team had experimented with SonarQube, a code quality and analysis tool. However, the lack of processes and resources has prevented further adoption so far. Additionally, the same team had considered introducing a vulnerability scanning tool, such as Snyk, into their CI/CD pipelines. The situation with SonarQube was reflected on in one interview as follows:

"...Introducing it (SonarQube) should not be a massive task, but if it finds something, there should be a workflow...”

Test automation is one way to address code security risks indirectly, as mentioned in table 3.2. The amount of software testing varied between projects. While the interview sessions did not allow for in-depth discussion on the subject matter, the information retrieved from the interviews indicate for improvements to be made in that area. The combination of the lack of formal processes and available resources can be attributed to the current state of software testing. One data engineer described their situation as follows:

“...It would be nice to do them (tests). Just need to find a light framework that could be used to simulate the *<anonymized data engineering product>* environment...”

Additionally, one software engineering project had to find creative ways to test some of their functionalities, as creating mock APIs to simulate physical environments was not enough to assure that code changes worked as intended.

Another way to mitigate code security risks is by avoiding the exposure of credentials, as mentioned in table 3.2. While none of the interviewed teams were using pre-commit hooks to scan for leaking sensitive information, more than one team used key vault services, which provide a secure way to access secrets from within the application.

In summary, managing the CI/CD pipeline is much more complex compared to conducting code reviews. The current state of securing the source code suggests that Neste needs to take a step back and introduce formal processes and guidelines for teams to apply for a more standardized release cycle, before proceeding with introducing expensive application security tools in the workflows, which might not even fit in a large number of projects.

Open source code dependency management

Software supply chain attacks have highlighted the need for stricter controls regarding dependency management in software projects. However, the general answer in the interviews on this topic was, that it was up-to developers’ own discretion to utilize third party libraries within their projects. However, one team revealed that they follow a change management process in form of a checklist for introducing new dependencies. The checklist involves a review from Neste’s security team. Once the new solution has been reviewed and considered safe, it can be used in the project. The process was not seen as perfect and one interviewee commented on it as follows:

“...This handover checklist is not quite a rigid process, and I would feel better if it would be more strictly defined. And then also, more strictly defined on who are actually the people authorized to provide this security approval..."

Additionally, answer from another interview indicates that there is some awareness regarding the legal risks arising from introducing dependencies with possible license issues:

"...Of course, you have to take a look at the license, because you can not import just anything into commercial environments..."

6 Discussion

This chapter gives answers to the research questions defined earlier in this thesis. Additionally, the results are compared to related studies, which are followed up by discussion on possible limitations related to the study.

6.1 Answers to research questions

RQ1: What are the commonly adopted practices for secure software development today?

To answer this research question, a literature review was conducted. Modern software development revolves around the DevOps paradigm, which focuses on eliminating friction between development and operations teams through a culture of collaboration and automation-focused software development practices. One of the key benefits of adopting DevOps is achieving faster development cycles. While adopting DevOps increases agility, it also creates additional needs to ensure the security aspects of the software.

DevSecOps is a principle that builds on top of DevOps, while prioritizing security. *Shifting security to the left* is one of the key concepts in DevSecOps. It involves integrating security measures from the beginning of the software development process. A typical example involves scanning for potential vulnerabilities within the source code artifacts in the CI/CD pipeline. Similar to DevOps, DevSecOps is far more than just introducing tools in the development workflows. Successfully adopting DevSecOps requires significant organizational efforts to create a culture of inter-team collaboration and cross-functionality between the development, security, and operations teams, in order for tools and practices to be effective.

At the same time, the evolution of how modern software is based on bits and pieces of third party code today has developed a whole new set of challenges that threaten the integrity of *software supply chains*. With more and more software supply chain attacks occurring today, teams must pay attention on introducing processes that focus on mitigating risks originating from putting too much trust in their development activities, such as code reviews and management of open source components found within their software. Creating appropriate solutions to solve these issues is an ongoing industry effort, but teams being aware of ways to manage their code dependencies, and concepts such as *software bill of materials*, can work as the starting point in the journey of improving security of their software beyond the CI/CD pipeline.

RQ2: How do Neste's software development teams currently address the state of code security in their projects?

To answer this research question, a qualitative case study was conducted. The answers were split into three categories: *Code reviews*, *Code security in the CI/CD pipeline*, and *Open source code dependency management*. With limited technical guidance from Neste, it is difficult to measure the maturity level of these practices within the interviewed teams. For example, while teams are using code reviews to assess code quality, a typical code review is a manual effort and there is no unified standard on what to review. However, due to the presence of code reviews across the interviewed teams' workflows, Neste could introduce guidelines and processes for teams to follow in a relatively near future.

The same can not necessarily be said for introducing application security tools in the CI/CD pipeline. One reason is that the tools do not necessarily support every type of code artifact. This is more evident with data engineering projects, where the code artifacts are connected to proprietary technologies. On the other hand, lack of knowledge, resources, and formal processes also prevent teams from adopting these tools. Moreover, the current state of software testing, combined with other ways to indirectly address code security (e.g., management of sensitive information), indicate that standardizing these steps should take priority before introducing expensive code scanning tools, which might bring teams' workflows to a halt instead.

The risks existing in software supply chains today, combined with the answers retrieved from the interviews suggest that Neste should take actions to ensure the open source software used within their projects is technically and legally sound. Answers from one interview indicate that Neste has a formal process to review open source components, but is not in broad use and the responsibility is mostly left in developers' hands.

RQ3: Which code security measures should be integrated into existing Neste's software development projects?

Based on the findings from the previous two research questions, it is possible to introduce tools and processes to help developers build more secure software, albeit in limited capacity. In short term, to ensure business continuity in regards to legal risks arising from possible license conflicts with open source software, there should be a company-wide policy on what licenses are, and are not allowed to be used within custom software. Additionally, teams should consider adopting products such as *Open source vulnerabilities* (OSV), pictured in figure 6.1, to check for vulnerabilities within dependencies, while also respecting the intellectual property rights by not submitting the source code to third parties. An example vulnerability triage workflow consists of three steps (Chang et al., 2021). First, the open source user sends a query to the OSV API, containing basic information of the target dependency or a relevant git commit hash. Then, the API responds by aggregating vulnerability data from vulnerability databases using the osv schema. Finally, the user can then analyze the response to perform potential security fixes.

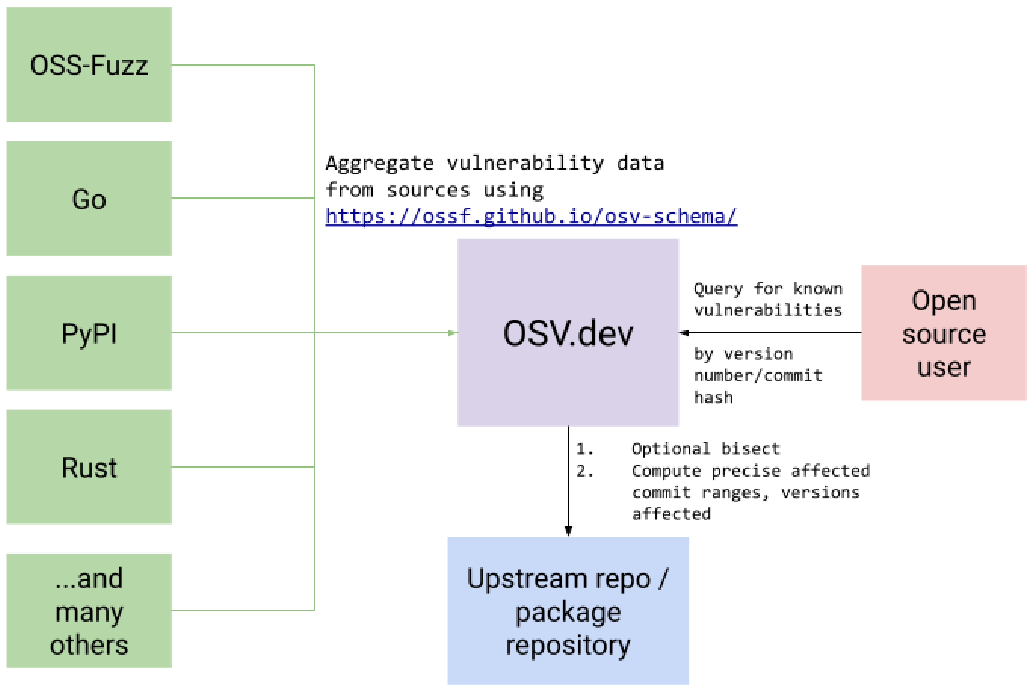
OSS-Fuzz
Go
PyPI
Rust
...and many others
Aggregate vulnerability data from sources using https://ossf.github.io/osv-schema/
OSV.dev
Query for known vulnerabilities
by version number/commit hash
Open source user
1. Optional bisect
2. Compute precise affected commit ranges, versions affected
Upstream repo / package repository

Figure 6.1: Workflow diagram for Open source vulnerabilities project (OSV Team, 2022).

Services such as OSV could be directly integrated into development projects' CI/CD pipeline, which could trigger alerts to Neste's cybersecurity team based on the severity of the issue. As mentioned in chapter 4, development teams can stay secure by simply patching the vulnerable code dependency in majority of disclosed OSS vulnerabilities. Therefore, the security incident management process should be relatively straightforward in most situation. However, this approach requires some changes to which stakeholders are directly involved with software development projects from Neste's side.

Code reviews are another topic to which Neste can have an effect on in a relatively near future due to their presence in the workflows of the interviewed teams. In theory, one of the easier solutions would be to choose a suitable software development project and create a workflow for code reviews with the help of a commercial vendor's consulting team providing necessary technical assistance. The problem with this approach is that the workflows and standards might be difficult to generalize across various types of software development. Therefore, the initial focus could be pointed towards creating formal processes for manual testing of changes in standardized pre-production environments. The requirements for this workflow are roughly as follows:

- Code deployment environments (e.g., development/staging/production) should be provisioned and configured with Infrastructure-as-code. This approach would allow to test changes in a more reliable manner within pre-production environments.

- Pull requests should automatically push changes to the next appropriate environment according to the selected branching strategy. For example, approving changes in Dev branch would push changes to Staging environment. In light of code reviews being a possible entry point for a software supply chain attack, two separate individuals should review and approve the changes. If possible to implement, a *CODEOWNERS* file should help to automatically assign most suitable code reviewers.

- The manual testing process should include a checklist. The research done in this survey is insufficient to explain what the contents of one should include. Therefore, Neste could take a quantitative research approach and create a survey to collect data on conducting code reviews from individual developers.

On one hand, integrating commercial code vulnerability scanning tools into the CI/CD pipeline is currently not a cost-efficient strategy according to answers to RQ2. On the other hand, automated tests are an integral part of checking that software under development stays working correctly (Vocke, 2018). They also address code security risks indirectly, as mentioned in table 3.2. Therefore, a standardized approach to test automation is something that should be considered. However, it is not feasible to introduce a complete plan for software testing in this thesis.

Currently, some actions can be taken to consider the security of source code itself. Neste has facilitated ongoing workshops to support this study. So far, one suggestion has been to create a catalogue of *template repositories* for projects to start from. These repositories could be pre-configured to automatically support previously mentioned two recommendations, but could also include at least the following suggestions from the viewpoint of code security:

- Requirement for commits to be digitally signed to prove the origin of the individual behind the code artifact change.
- Prevent sensitive information from being leaked by including a secrets management solution by default within the template repository.

6.2 Related studies

Neste's reliance on almost completely outsourcing software development to third parties has resulted in a rather novel research problem in the context of secure software development. Other academic studies that also aimed to identify challenges related to DevSecOps, were able to achieve more technical results. This could be explained by closer access to technical stakeholders from a researcher's standpoint. Nevertheless, these studies can show what Neste could potentially expect when going forwards with more comprehensive experiments in the future.

One example of a somewhat related study is a quantitative case study, which focused on identifying and documenting experiences of various teams' journey on implementing DevSecOps (Colliander, 2022). The findings suggest that the focus should be on allocating sufficient time for developers and security personnel to improve on their security skills, instead of needlessly spending time on selecting the most fitting tools (Colliander, 2022). The questionnaire used in the study is quite comprehensive and parts of it could be reused in a future Neste software development survey.

Another study had integrated commercial DevSecOps tools into existing workflows of an actual software development team (Riski, 2022). According to that study, a code scanning tool is more likely to be adopted if it is able to present results to developers within the existing context of the development process (Riski, 2022). Another observed obstacle was the immaturity of Integrated Development Environment (IDE) integrations for some of the code scanning tools used in the study (Riski, 2022). The research done in the study by Riski could include potential future research topics at Neste, as it contains relevant information to the research question RQ3.

6.3 Limitations

There are some concerns regarding the validity of this thesis, for both literature review and the case study. The research protocol for the literature review done in this thesis is not as thorough as in studies utilizing systematic literature review as the primary research method. Additionally, *DevSecOps*, and in particular *software supply chains*, are emerging concepts in software development. This means that finding peer-reviewed literature is difficult, and thus grey literature was used quite extensively to explain parts of these topics. However, developer-first security is a constantly evolving field, which is why it might not even be feasible to conduct systematic literature reviews to keep up with the latest developments.

On the other hand, conducting the case study had its own challenges. Neste's company structure meant that finding technical guidance for this thesis was not straightforward due to the general lack of software developers being directly employed by the company. Moreover, the four teams participating in the interviews do not necessarily represent the state of software development as a whole at Neste. Additionally, due to lack of time and other resources, teams developing software through other means such as Integration development and Robot process automation, were not included in this case study. In hindsight, action research could have been a more appropriate research method for this thesis, as interviewing stakeholders from multiple teams allowed for a broader view at the software development practices, at the expense of immediately actionable results in this situation.

Finally, contents of both interview formats were purposefully broad. This is due to to initial interviews beginning before the final thesis direction was selected. It is also one of the reasons for the usage of two different interview formats to collect data. However, the interview formats A and B are similar enough for data categorization to be possible without overwhelming obstacles. If there was a need to ask for additional clarifications, we used emails and direct chats to fill in the missing information, and thus align insights between the interview formats. Furthermore, the interview formats were imperfect, and some topics such as monitoring need further investigation before making any company-wide policies on that matter.

7 Conclusions

The modern way of rapidly developing software has introduced security gaps that can be exploited by third parties. Vulnerability scanning tools are designed to detect issues within custom code. However, the trust that software developers have put into third party code, has been abused by malicious actors. The emerging attack vectors are able to bypass some of the modern application security measures, which has already resulted in major consequences in the software industry. This has resulted in industry effort to mitigate the risks of careless utilization of open source components, on which the vast majority of modern software is built upon.

This thesis was commissioned by Neste Corporation and focused on investigating the current state of how their outsourced software development teams address code security. We conducted a qualitative case study by interviewing a variety of stakeholders from four different teams. In summary, code security is currently not at the forefront, but modern software development tools and technologies are used for agile software development across projects, which means that security improvements can be done. This thesis proposes a small set of suggestions that address code security and encourage collaboration between stakeholders. The primary suggestion is to ensure that open source components are reviewed for vulnerabilities and their licenses allow them to be used within closed source projects.

In hindsight, more actionable results could have been achieved with a more involved research method such as action research. Nevertheless, this thesis reveals some opportunities for future research. From Neste's point of view, the next steps should include active monitoring and formalization of the newly suggested open source security enhancement process, preferably by leveraging automation instead of manually checking on individual projects' statuses. However, a pre-requisite for this involves creating company-wide guidelines for CI/CD pipeline configurations, at a reasonably abstract level. From a broader viewpoint, one of the more interesting opportunities for future research could be conducting more practical research by programmatically consuming contents of a software bill of materials for a variety of use cases.

Bibliography

Arvanitis, I., Ntousakis, G., Ioannidis, S., and Vasilakis, N. (2022). "A Systematic Analysis of the Event-Stream Incident". In: *Proceedings of the 15th European Workshop on Systems Security.* EuroSec '22. Rennes, France: Association for Computing Machinery, pp. 22–28. ISBN: 9781450392556. DOI: 10.1145/3517208.3523753. URL: https://doi.org/10.1145/3517208.3523753.

Bell, L. (2022). *Why we need to stop shifting cyber security left.* [Online; accessed June 2, 2022]. URL: https://blog.safestack.io/secure-development-stop-shifting-cyber-security-left.

Bird, J. (2015). *Continuous Delivery versus Continuous Deployment.* [Online; accessed June 5, 2022]. URL: http://radar.oreilly.com/2015/10/continuous-delivery-versus-continuous-deployment.html.

Black, P., Okun, V., and Guttman, B. (Oct. 2021). *Guidelines on Minimum Standards for Developer Verification of Software.* en. [Online; accessed June 16, 2022]. DOI: https://doi.org/10.6028/NIST.IR.8397. URL: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=933350.

Blit, R. (2022). *What Is SLSA? SLSA Explained In 5 Minutes.* [Online; accessed July 14, 2022.] URL: https://www.legitsecurity.com/blog/what-is-slsa-slsa-explained-in-5-minutes.

Carmody, S., Coravos, A., Fahs, G., Hatch, A., Medina, J., Woods, B., and Corman, J. (Feb. 2021). "Building resilient medical technology supply chains with a software bill of materials". In: *npj Digital Medicine* 4, p. 34. DOI: 10.1038/s41746-021-00403-w.

Caum, C. (2013). *Continuous Delivery Vs. Continuous Deployment: What's the Diff?* [Online; accessed June 5, 2022]. URL: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff/.

Chang, O., Lewandowski, K., and Google Security Team (2021). *Launching OSV - Better vulnerability triage for open source.* [Online; accessed September 19, 2022.] URL: https://opensource.googleblog.com/2021/02/launching-osv-better-vulnerability.html.

Colliander, C. (2022). "Challenges of DevSecOps". English. Master's thesis. University of Helsinki, Faculty of Science. URL: http://hdl.handle.net/10138/342887.

Cook, K. (2021). *An emergency re-review of kernel commits authored by members of the University of Minnesota, due to the Hypocrite Commits research paper.* [Online; accessed June 13, 2022.] URL: https://lore.kernel.org/lkml/202105051005.49BFABCE@keescook/.

Enck, W. and Williams, L. (Mar. 2022). "Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations". In: *IEEE Security & Privacy* 20.2, pp. 96–100. ISSN: 1558-4046. DOI: 10.1109/MSEC.2022.3142338.

Ensor, M. and Stevens, D. (2021). *Shifting left on security: Securing software supply chains.* [Online; accessed June 1, 2022]. URL: https://cloud.google.com/files/shifting-left-on-security.pdf.

Executive Office of the President (May 2021). *Executive Order 14028 on Improving the Nation's Cybersecurity.* [Online; accessed June 5, 2022]. URL: https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity.

Fitzgerald, B. and Stol, K.-J. (2014). "Continuous Software Engineering and beyond: Trends and Challenges". In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering.* RCoSE 2014. Hyderabad, India: Association for Computing Machinery, pp. 1–9. ISBN: 9781450328562. DOI: 10.1145/2593812.2593813. URL: https://doi.org/10.1145/2593812.2593813.

Fowler, M. (2006). *Continuous Integration.* [Online; accessed June 5, 2022]. URL: https://martinfowler.com/articles/continuousIntegration.html.

Grady, J. and DeMattia, A. (July 2021). *Reaching the Tipping Point of Web Application and API Security.* en. [Online; accessed June 5, 2022]. URL: https://assets.ctfassets.net/6pk8mg3yh2ee/2ROnWrnlNeMekgOgvb4JmO/004ef415e2bbd4c1f5939cdd676518e7/ESG-Research-Insights-Paper-Fastly-Web-App-and-API-Protection-July-2021_English_FINAL.pdf.

Grander, D. and Tal, L. (2018). *A post-mortem of the malicious event-stream backdoor.* [Online; accessed June 13, 2022.] URL: https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/.

Holz, T. and Oprea, A. (2021). *IEEE S&P'21 Program Committee Statement Regarding The "Hypocrite Commits" Paper.* [Online; accessed July 18, 2022.] URL: https://www.ieee-security.org/TC/SP2021/downloads/2021_PC_Statement.pdf.

Jaatun, M. G. and Soares Cruzes, D. (June 2021). "Care and Feeding of Your Security Champion". In: *2021 International Conference on Cyber Situational Awareness, Data*

Analytics and Assessment (CyberSA), pp. 1–7. DOI: 10.1109/CyberSA52016.2021.9478254.

Kaczorowski, M. (Sept. 2020). *Secure at every step: What is software supply chain security and why does it matter?* [Online; accessed September 12, 2022]. URL: https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/.

Lewandowski, K. and Lodato, M. (2021). "Introducing slsa, an end-to-end framework for supply chain integrity". In: *Google Online Security Blog*. [Online; accessed June 12, 2022.] URL: https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html.

Myrbakken, H. and Colomo-Palacios, R. (Sept. 2017). "DevSecOps: A Multivocal Literature Review". In: pp. 17–29. ISBN: 978-3-319-67382-0. DOI: 10.1007/978-3-319-67383-7_2.

Narayan, S., Disselkoen, C., Garfinkel, T., Froyd, N., Rahm, E., Lerner, S., Shacham, H., and Stefan, D. (Aug. 2020). "Retrofitting Fine Grain Isolation in the Firefox Renderer". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, pp. 699–716. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

Ohm, M., Plate, H., Sykosch, A., and Meier, M. (2020). "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by C. Maurice, L. Bilge, G. Stringhini, and N. Neves. Cham: Springer International Publishing, pp. 23–43. ISBN: 978-3-030-52683-2.

OSV Team (2022). *Open Source Vulnerabilities Github repository*. [Online; accessed August 23, 2022.] URL: https://github.com/google/osv.dev.

OWASP CycloneDX (2022). *CycloneDX informational website*. [Online; accessed September 8, 2022]. URL: https://owasp.org/www-project-cyclonedx/.

Palafox, J. (2022). *Achieving SLSA 3 Compliance with GitHub Actions and Sigstore for Go modules*. [Online; accessed July 14, 2022.] URL: https://github.blog/2022-04-07-slsa-3-compliance-with-github-actions/.

Pasha, M. (2021). *WAF vs. RASP: A Comparison and Guide to Leveraging Both*. [Online; accessed June 2, 2022. Publication date retrieved by inspecting html source code]. URL: https://www.traceable.ai/blog-post/waf-vs-rasp-a-comparison-and-guide-to-leveraging-both.

Plutora (2019). *DevSecOps diagram.* [Online; accessed June 1, 2022]. URL: https://1ohvy81v7br01wtgnj4bf0ek-wpengine.netdna-ssl.com/wp-content/uploads/2019/03/DevSecOps-Diagram.png.

Rajapakse, R. N., Zahedi, M., Babar, M. A., and Shen, H. (2022). "Challenges and solutions when adopting DevSecOps: A systematic review". In: *Information and Software Technology* 141, p. 106700. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106700. URL: https://www.sciencedirect.com/science/article/pii/S0950584921001543.

Rajapakse, R. N., Zahedi, M., and Babar, M. A. (2021). "An Empirical Analysis of Practitioners' Perspectives on Security Tool Integration into DevOps". In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. New York, NY, USA: Association for Computing Machinery. ISBN: 9781450386654. URL: https://doi.org/10.1145/3475716.3475776.

RedHat (2022). *What is a CI/CD pipeline?* [Online; accessed June 5, 2022]. URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline.

Riski, T. (2022). "Challenges in Realizing DevSecOps: A Case Study". English. Master's thesis. Aalto University. School of Science. URL: http://urn.fi/URN:NBN:fi:aalto-202203272602.

Runeson, P. and Höst, M. (2009). "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14.2, pp. 131–164.

Senapathi, M., Buchan, J., and Osman, H. (2018). "DevOps Capabilities, Practices, and Challenges: Insights from a Case Study". In: *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. EASE'18. Christchurch, New Zealand: Association for Computing Machinery, pp. 57–67. ISBN: 9781450364034. DOI: 10.1145/3210459.3210465. URL: https://doi.org/10.1145/3210459.3210465.

Shahin, M., Ali Babar, M., and Zhu, L. (2017). "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5, pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

Shahin, M., Zahedi, M., Babar, M. A., and Zhu, L. (Sept. 2018). "An empirical study of architecting for continuous delivery and deployment". In: *Empirical Software Engineering* 24.3, pp. 1061–1108. DOI: 10.1007/s10664-018-9651-4. URL: https://doi.org/10.1007/s10664-018-9651-4.

SLSA (2022b). *Software supply chain levels.* [Online; accessed June 12, 2022. 'Last updated' year retrieved from the SLSA GitHub repository.] URL: https://slsa.dev/spec/v0.1/terminology.

– (2022c). *Software supply chain levels.* [Online; accessed July 15, 2022. 'Last updated' year retrieved from the SLSA GitHub repository.] URL: https://slsa.dev/spec/v0.1/#supply-chain-threats.

– (2022a). *Software supply chain levels.* [Online; accessed July 15, 2022. SLSA Version 0.1]. URL: https://slsa.dev/.

– (2022d). *Software supply chain levels.* [Online; accessed July 15, 2022. 'Last updated' year retrieved from the SLSA GitHub repository.] URL: https://slsa.dev/spec/v0.1/levels.

Smeds, J., Nybom, K., and Porres, I. (2015). "DevOps: A Definition and Perceived Adoption Impediments". In: *Agile Processes in Software Engineering and Extreme Programming.* Ed. by C. Lassenius, T. Dingsøyr, and M. Paasivaara. Cham: Springer International Publishing, pp. 166–177. ISBN: 978-3-319-18612-2.

SonaType (2021). *The 2021 State of the Software Supply Chain Report.* [Online; accessed June 5, 2022]. URL: https://www.sonatype.com/resources/white-paper-2021-state-of-the-software-supply-chain-report-2021.

Synopsys (2022). *The 2022 Open Source Security and Risk Analysis report.* [Online; accessed July 9, 2022.] URL: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf.

Synopsys editorial team (2016). *Guide to open source licenses.* [Online; accessed July 18, 2022.] URL: https://www.synopsys.com/blogs/software-security/open-source-licenses/.

Vocke, H. (2018). *The Practical Test Pyramid.* [Online; accessed August 23, 2022]. URL: https://martinfowler.com/articles/practical-test-pyramid.html.

Wegner, V. (2020). *DevSecOps basics: 9 tips for shifting left.* [Online; accessed June 2, 2022]. URL: https://about.gitlab.com/blog/2020/06/23/efficient-devsecops-nine-tips-shift-left/.

WhiteSource (2020). *The State of Open Source Security Vulnerabilities - WhiteSource Annual Report 2020.* [Online; accessed September 12, 2022. WhiteSource has been rebranded as Mend in 2022.] URL: https://www.mend.io/wp-content/media/2020/03/Annual_Report_2020_12.03.20.pdf.