Dependency Testing: a critical aspect in DevSecOps

Aïssata Maiga, maiga@kth.se
Pontus Cowling Mantefors, pontuscm@kth.se
DD2482

April 28, 2022

1 Introduction

This report will introduce you to the concept of dependency testing as a part of DevSecOps and explain to you its urgency together with possible solutions. Two popular, free and open source tools have been tested and this is presented in section 4.2.

1.1 DevSecOps == DevOps + Security

DevSecOps is the concept of bringing security practices into the practices of DevOps. While development cycles in the past could last for months or years, it might not have been so problematic separating the security work to the final stages of development. Today, development is very often done applying DevOps processes, including shorter cycles and CI/CD pipelines. Keeping security separated in these kinds of processes risks creating bottlenecks in the development. This has created the need to include the security concerns into the DevOps processes to be a shared responsibility throughout the entire cycle.

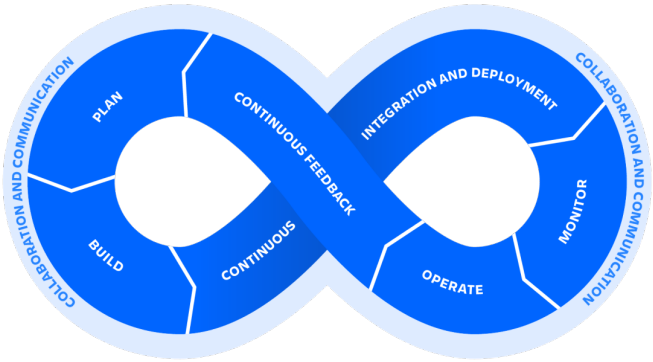

Figure 1: DevOps cycle [13]

1.2 Dependency checking

One of the biggest challenges in DevSecOps is dependency testing [2]. With software including more and more dependencies it has become a growing and urgent matter with new security threats surfacing on a regular basis.

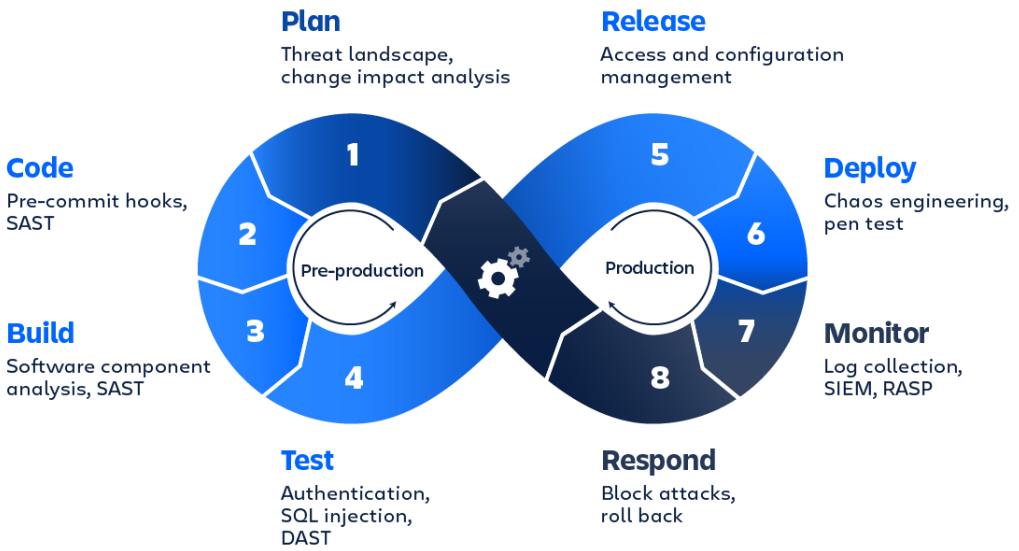

Figure 2: DevSecOps cycle [13]

Examples of this include one developer intentionally sabotaging open source libraries with 22 M downloads per week rendering a very large number of projects useless [8]. Another was when the Node.js library "event-stream" with 2 M downloads per week was infected with malicious code attempting to steal bitcoin[7].

2 DevSecOps

2.1 Why DevSecOps?

With the adoption of the Agile development cycle as well as CI/CD, software is developed much faster - this requires automation to not introduce vulnerabilities at every step of this accelerated process. Security cannot be delayed to the last moment. This increases the burden on security teams. In addition, this increases the risk of having severe bugs slipping into production which much more costly. According to NIST (National Institute of Standards and Technology) the cost of fixing bugs in production can be 30 times higher if discovered late [11]. Other arguments are faster deployment,

reducing "significant [...] costs in end-user experience and satisfaction, loss of revenues, and brand damage" [4].

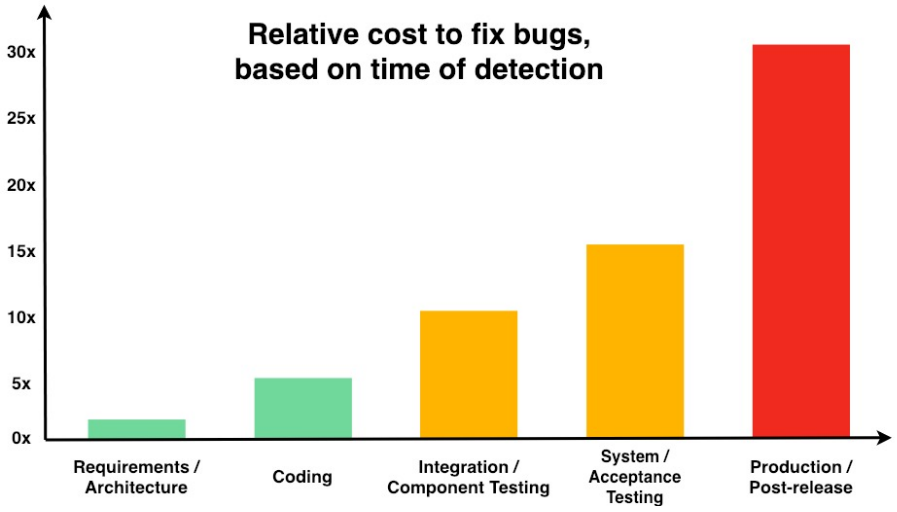

Figure 3: NIST: bugfixing costs [11]

2.2 Cultural shift

Several sources highlight the importance of a cultural shift to completely adopt DevSecOps procedures.

In their multivocal study of DevSecOps, Myrbakken and Colomo-Palacios [6] define a DevSecOps culture as: promoting "a culture where operations and development also work on integrating security in their work [...] involving the security team from the planning stages, and making sure everyone agrees that security is everyone's responsibilities".

DevSecOps must become a foundational step of software development, and security an "an integral and continuous component of the application lifecycle". This requires organisational change and other active processes to make the security team a pivotal part of any project [10].

This cultural shift is challenging, and Mao et al. identified several difficulties [5]. Internal challenges are "culture resistance, high cost and solidified organizational structure", while major external challenges are: "lack of DevSecOps experts, tools, and mature DevSecOps solutions".

2.3 Methods and best practices

Integration of security throughout the development cycle is the most robust. Still in grey literature, we found 5 methods. [4] [12].

1. Static application security testing (SAST)
2. Dynamic security application testing (DAST):
3. Interactive application security testing (IAST)
4. Source composition analysis (SCA)
5. End-to-end scanning of containers

This report focuses only on SCA which is the study of dependencies with tools such as Snyk or OWASP presented in section 4.2.

3 The problem

With effective software reuse, advocated in recent years in software engineering, comes the problem of bloated dependencies. Soto Valero et al., studied bloated dependencies [3], that is, additional software that is a part of the compiled code but are of no use in the program. To quantify the problem, the researchers found that 75% of Maven dependencies (the code base analysed in their study) are bloated. They also stress that dependency management is a concern of the community, and a majority of PR's (pull requests), specifically 21/26 are trying to remove bloated dependencies. When Soto Valero et al., made a contribution, 16/17 of the PR's were accepted and merged.

In Black Duck Software (2017) audit, [12] over 1,000 commercial applications showed that 96% included OSS components. 60% of applications contained known security vulnerabilities, with some known over 4 years. In that audit, less than a third of companies had procedures in place to track and remove vulnerabilities.

Considering these large numbers of vulnerable dependencies that are not dealt with or sometimes are even unknown, it is easy to see that making sure that security is withheld becomes a very hard task. This points the light to the need for dependency testing.

In the chapter below, we will describe how dependency checking is usually carried out to find vulnerabilities, and solutions currently in place. We will see that when not automated, those solutions can be rather cumbersome.

4 Solution

4.1 Checking against databases

According to The State of Open Source Security 2020 [9], OSS continues to expand, with npm increasing by a third in 2019 (1,300,000 packages in 2020). This lead to open source vulnerabilities "discovered in indirect dependencies":

- npm – 86%
- Ruby – 81%
- Java – 74%

According to the CVE (Common Vulnerabilities and Exposures) [1], "A flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components.".

As Soto-Valero et al., put it in their paper, "software is bloated". There exist tools to find vulnerabilities, analyse the versions in your dependencies in your codebase, and match them against known vulnerabilities. Two examples are the NVD database or the CVE database.

The NVD is a database maintained by the U.S. government while the CVE is open-sourced. The NVD database uses the CVE in addition to its own. There are to date 173 917 CVE records reporting vulnerabilities. The existence of those databases enable automation in fixing vulnerabilities, and publishing flaws in software. Depending on the severity of your vulnerability, a score, ranging from 0 to 10 will be attributed. A good practice in DevOps is to locally control the state of vulnerabilities in a CI/CD pipeline, and set it to fail if the CVE score is too high [2].

4.2 Existing tools

Two existing tools are Snyk and OWASP Dependency-Check. Snyk seems popular and is recommended by many resources when trying to improve security in CI/CD. OWASP is a global foundation that has worked for two decades with the goal of improving the security of software and their

Dependency-Check is a well maintained tool that is often included and recommended when dependency testing tools are listed by security websites, blogs, etc.

4.2.1 Snyk

This part will analyse the offer from Snyk as well as the experience in using it.

Snyk site proposes to "Find and automatically fix vulnerabilities [...], open source dependencies, containers [...] — all powered by Snyk's industry-leading security intelligence.". We are left wondering what this "intelligence" may refer to as this kind of service is supposed to check against CVE databases.

Snyk proposes a CLI tool, a plugin for VS code as well as a UI. The UI can very easily be connected to Github through the settings. Then, there is an option to scan all public repositories. In the context of this essay, I chose a website built with Gatsby (a SSG) with many dependencies.

Regarding languages, Snyk proposes to fix dependencies for four main languages: JavaScript, Python, .NET and PHP. It seems like a reasonable focus as those languages are mostly used for the web, where the problem of bloated/outdated/ outright dangerous dependencies is the most prevalent.

After registering to Snyk service, one can login to her/his dashboard and find information about the state of the repository/ies. The state of the JavaScript repository left much to be desired, and Snyk found a whopping 549 vulnerabilities in the dependencies, ranging from critical to low.

The service allows you to open a PR from the Dashboard, or automatically. Checking the linked repository, Snyk had opened more than a hundred PR's to fix dependencies. The tool is not completely optimised as it can open several PR's for the same problem, as for example, bumping webassemblyjs from 1.8 to 1.11.

This means that the tool is not verifying that a PR was already opened to fix a vulnerability. It will keep building the site and report all *current* vulnerabilities.

This brings us to another problem with Snyk. To find vulnerabilities to fix, it is building the website in question. It can cause problems with other providers such as services hosting the website. In this case, Snyk consumed the build minutes for the period (30 days), using more than 50% of the allotted time. The immediate issue was that trespassing the allotted time would

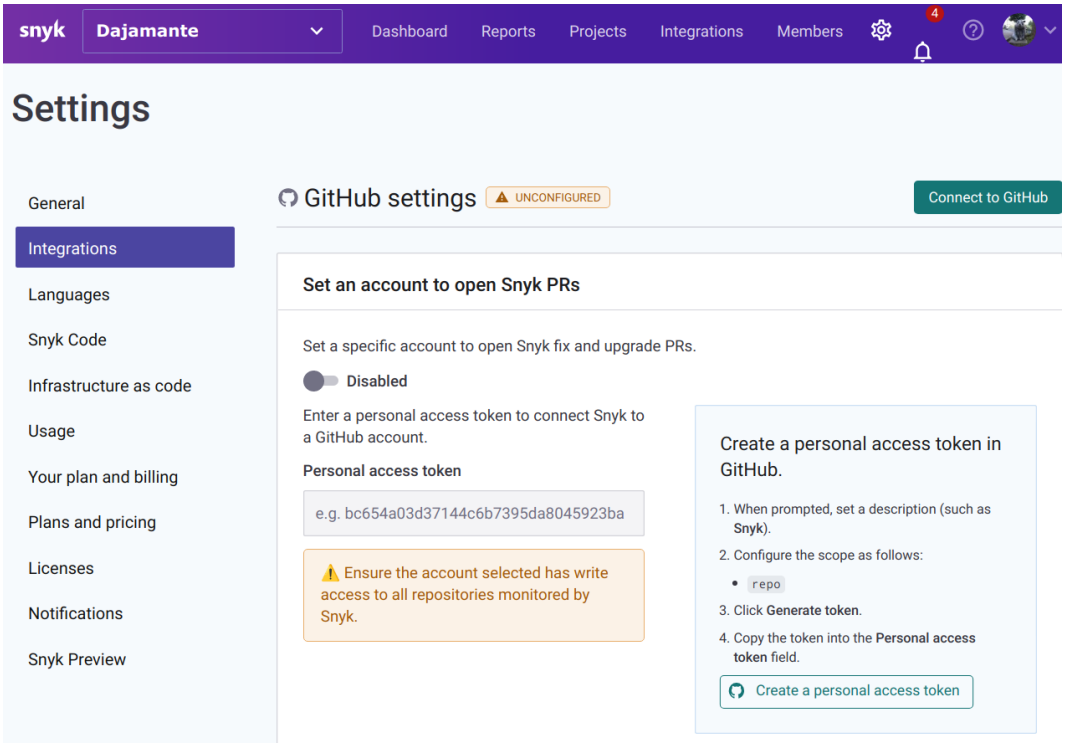

Figure 4: Snyk settings

lead the service provider to increase the extra build pack (automatically). The cost was $7 to go from 300 minutes to 500 minutes. This is not a major issue at the individual level, but in bigger teams, with bigger needs, the tool can cause unplanned spending.

To conclude, this is a great tool that compares your vulnerabilities against a CVE database. The UI is agreeable while the functioning can be perfected (check whether a PR already exists for this vulnerability, excessive builds).

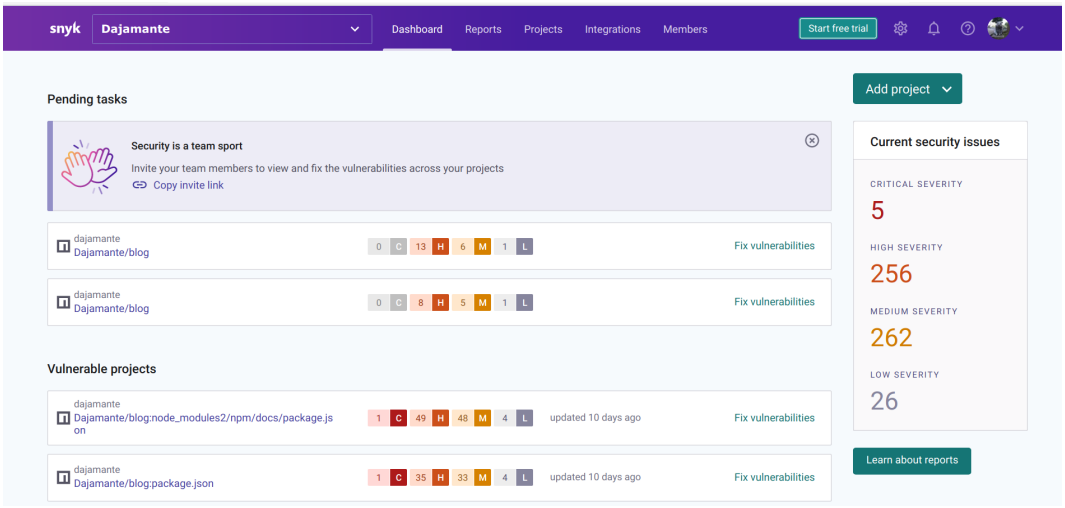

Figure 5: Pending tasks

Figure 8: Number of builds per day

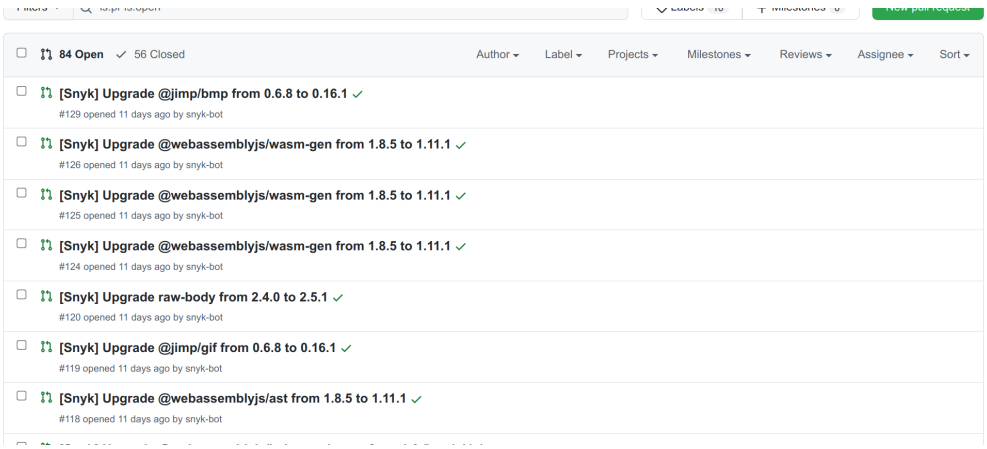

Figure 6: List of pull requests

4.2.2 OWASP Dependency-Check

OWASP Dependency-Check is an SCA that searches for vulnerabilities that are publicly disclosed in a number of databases. Included in the list of databases is the previously mentioned NVD but also other 3rd party services and data sources such as the NPM Audit API, the OSS index, RetireJS, and Bundler Audit are used. The tool is provided with a command line interface but also as a Maven, Jenkins, SBT or Gradle plugin as well as an Ant task. Once installed it updates itself automatically using "NVD Data Feeds" provided by NIST. OWASP Dependency-Check currently only offers full support for Java and .NET but it also has some experimental analysers for Python, Ruby, Node.js, CocoaPods, PHP, and Swift.

At the core of the tool is an engine that gathers information about all dependencies and tries to identify their Common Platform Enumeration (CPE). CPE is a standardised identification system used for IT systems, software, and packages. If a CPE is identified it is checked against the databases and services mentioned above and the CVE entries are listed in a report available in HTML, XML, CSV, JSON, JUnit and SARIF formats.

Installing Dependency-Check's command line interface is a quite simple procedure. It consists of just downloading a zip file from the official website and unzipping it in a desired location. After this there is a supplied shell script file that can be run as a terminal command. Running it without any

Figure 7: Build minutes per day

parameters lists the most common options. To analyse a repository one simply runs the program with -s and the local path to where you have placed your repository. The HTML (default) report will be placed in the bin folder.

Looking at the report, most parts of it are quite self-explanatory. OWASP does however also supply an online documentation explaining in more detail how to read the report. Looking at figure 9 below, one can see that it starts with general information about the project, showing things like version number, total number of dependencies and how many of them are deemed vulnerable.

Dependency-Check is an open source tool performing a best effort analysis of 3rd party dependencies; false positives and false negatives may exist in the analysis performed by the tool. Use of the tool and the reporting provided constitutes acceptance for use in an AS IS condition, and there are NO warranties, implied or otherwise, with regard to the analysis or its use. Any use of the tool and the reporting provided is at the user's risk. In no event shall the copyright holder or OWASP be held liable for any damages whatsoever arising out of or in connection with the use of this tool, the analysis performed, or the resulting report.

How to read the report | **Suppressing false positives** | **Getting Help: github issues**

Sponsor

Project:

Scan Information (show all):

- *dependency-check version*: 7.0.4
- *Report Generated On*: Fri, 22 Apr 2022 12:05:22 +0200
- *Dependencies Scanned*: 309 (152 unique)
- *Vulnerable Dependencies*: 4
- *Vulnerabilities Found*: 5
- *Vulnerabilities Suppressed*: 0
- ...

Figure 9: General information

This is then followed by a summary (figure 10) showing the vulnerable dependencies, expandable to show all dependencies if desired. In this summary the vulnerabilities are listed showing level of severity and at what confidence level they are deemed vulnerable together with names, ID's, package, as well as CVE and evidence counts.

Summary

Display: Showing Vulnerable Dependencies (click to show all)

Dependency	Vulnerability IDs	Package	Highest Severity	CVE Count	Confidence	Evidence Count
JabRef-100.0.0.tar: jackson-databind-2.13.1.jar	cpe:2.3:a:fasterxml:jackson-databind:2.13.1:*:*:*:*:*:*:* cpe:2.3:a:fasterxml:jackson-modules-java8:2.13.1:*:*:*:*:*:*:*	pkg:maven/com.fasterxml.jackson.core/jackson-databind@2.13.1	HIGH	1	Highest	44
JabRef-100.0.0.tar: postgresql-42.3.2.jar	cpe:2.3:a:postgresql:postgresql_jdbc_driver:42.3.2:*:*:*:*:*:*:*	pkg:maven/org.postgresql/postgresql@42.3.2	CRITICAL	1	Low	74
JabRef-100.0.0.tar: scala-library-2.12.8.jar	cpe:2.3:a:scala-lang:scala:2.12.8:*:*:*:*:*:*:*	pkg:maven/org.scala-lang/scala-library@2.12.8	MEDIUM	1	Highest	40
JabRef-100.0.0.tar: secret-service-1.0.0.jar	cpe:2.3:a:gnome:gnome:1.0.0:*:*:*:*:*:*:* cpe:2.3:a:gnome:gnome-keyring:1.0.0:*:*:*:*:*:*:* cpe:2.3:a:gnome:gnome_keyring:1.0.0:*:*:*:*:*:*:*	pkg:maven/de.swiesend/secret-service@1.0.0	HIGH	2	Low	29

Figure 10: Summary

Following the summary, each of the vulnerabilities are listed with more detailed descriptions and information. As shown in figure 11 and 12 the information includes such things as licence, evidence, related dependencies, identifiers (with confidence levels) and a whole section showing the published vulnerability information including an explanation in plain text, scores, references and specified software versions.

JabRef-100.0.0.tar: postgresql-42.3.2.jar

Description:

PostgreSQL JDBC Driver Postgresql

License:

BSD-2-Clause: https://jdbc.postgresql.org/about/license.html

File Path: /Users/pontus/Documents/KTH/DD2480_se/Assignments/DD2480_Group25-Assignment3/jabref/build/distributions/JabRef-100.0.0.tar/JabRef-100.0.0/lib/postgresql-42.3.2.jar
MD5: 97d32ed41973f9d7f49d0feb1fb9c662
SHA1: 8fd7a20f008a58b97b26ba5c5084ee61602203aa
SHA256:b9013e092d4a6dae404b9b2eaeaaec82e7583dd450911fdc1e436a5aa62a25cf

Evidence

Related Dependencies

Identifiers

- pkg:maven/org.postgresql/postgresql@42.3.2 (*Confidence*:High)
- cpe:2.3:a:postgresql:postgresql_jdbc_driver:42.3.2:*:*:*:*:*:*:* (*Confidence*:Low) suppress

Figure 11: Vulnerability description

Figure 12: Published vulnerabilities

In conclusion, OWASP Dependency-Check is simple to use with reports that are quite easy to understand. Considering that performing the analysis and generating the report is done in less than 20 seconds for approximately 300 dependencies it could be considered quite fast. Since it is free and possible to integrate with CI/CD pipelines using very common tools such as Gradle, Maven or Jenkins it should be considered a viable option to use for both small and large corporations when looking at implementing DevSecOps. Especially if there is currently no dependency check being done.

5 Conclusion

In this essay, we have presented an important problem related to DevSecOps, specifically the reliance on dependencies and the security risks related to that. This is a problem that is especially common and severe in open source software. We have also tested two tools, and found that even if one had a better user interface, the second had better functionality without hidden costs. It was enlightening to be able to test and adopt DevSecOps techniques, and considering the relative simplicity of implementing these techniques, their low cost together with the benefits it should be a naturally integrated part of all DevOps based software development. As for us, the authors of this report, we plan on implementing dependency checking of software as part of our DevOps processes and routines.

References

[1] [n. d.] CVE. (). Retrieved 04/28/2022 from https://www.cve.org/.

[2] [n. d.] DevSecOps tools: what tools to apply good security practices. en-gb. (). Retrieved 04/28/2022 from https://www.padok.fr/.

[3] Soto Valero et al. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. eng. *Journal of Empirical Software Engineering*, 26, 3. Retrieved 04/28/2022 from http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-293389.

[4] [n. d.] Five Ways to Integrate Security with DevSecOps Tools. (). Retrieved 04/02/2022 from https://www.checkpoint.com/.

[5] Runfeng Mao. 2020. Preliminary Findings about DevSecOps from Grey Literature. In.

[6] Håvard Myrbakken and Ricardo Colomo-Palacios. 2017. DevSecOps: A Multivocal Literature Review. en. In Cham, 17–29. ISBN: 978-3-319-67383-7.

[7] [n. d.] Rogue Developer Infects Widely Used NodeJS Module to Steal Bitcoins. (). Retrieved 04/28/2022 from https://thehackernews.com/.

[8] Emma Roth. 2022. Open source developer corrupts widely-used libraries, affecting tons of projects. (2022). Retrieved 04/02/2022 from https://www.theverge.com/.

[9] [n. d.] Snyk | Develop fast. Stay secure. (). Retrieved 04/28/2022 from https://go.snyk.io/SoOSS-Report-2020.html.

[10] [n. d.] Successful DevSecOps begins with a cultural shift. (). Retrieved 04/02/2022 from https://www.coalfire.com/.

[11] [n. d.] The exponential cost of fixing bugs. (). Retrieved 04/21/2022 from https://deepsource.io/.

[12] Jaikumar Vijayan. [n. d.] 6 DevSecOps best practices: Automate early and often. en. (). Retrieved 04/02/2022 from https://techbeacon.com/.

[13] [n. d.] What is DevOps? en. (). Retrieved 04/28/2022 from https://www.atlassian.com/.