Mitigating Supply Chain Attacks through Detection of High-Risk Software Dependencies

Jana Vojnović

March 2023

Supervisors: Dr. ir. Erik Poll, Tess Sluijter-Stek, Leendert Brouwer

Executive summary

Open-source libraries are widely used in software development, as they allow developers to reuse code and reduce their workload. However, these dependencies can also pose security risks, since they can be incomplete, contain vulnerabilities or malware, that attackers exploit to perform malicious actions. These attacks, which are performed through open-source libraries, are called supply chain attacks and in the past few years, they have become extremely popular with a staggering 742% growth in 2022 [50].

This study aims to detect supply chain attack categories not captured by typical software composition analysis (SCA) tools. Consequently, it also aims to develop mitigations that would decrease the overall supply-chain attack surface.

Firstly, seven possible attack categories are identified and analyzed, leading to the creation of markers for each category. Markers are specific thresholds or characteristics that are used to detect each attack category. Then, a toy application that intentionally uses these high-risk dependencies from each category was created, and tested against an SCA tool Nexus IQ. The results showed that Rabobank's utilization of Nexus IQ only identifies vulnerable packages, while the addition of Nexus Firewall and Repository would provide protection against packages that contain malware (three categories; typosquatting, dependency confusion, and hostile takeover). However, our analysis revealed that there is no protection against abandoned and insufficiently maintained packages, and that there is potential for implementing additional checks to identify hostile takeover and malicious packages.

As a result of these limitations, we were inspired to create our own tool that is capable of calculating a health score for any open-source package found on PyPi, npm, Maven or conda. Additionally, it can also flag packages on certain characteristics and provide according warnings to the user. These warnings can pinpoint four attack categories: abandoned packages, insufficiently maintained packages, hostile takeover through maintainer domain hijack, and certain malicious packages (very fresh or one-version packages). The score is calculated through 15 markers which are collected from Libraries.io API and GitHub API. The tool can be used in two ways; as a desktop application using GUI, or as a web application using REST API. After conducting successful tests, we confidently claim that this tool performs very well and represents a valuable addition to Nexus IQ, in terms of protecting developers against supply chain attacks.

Finally, we strongly believe that enhancing repository security through stronger namespace convention, malware detection and integration of SCA tools, as well as compensating maintainers of essential projects, would greatly diminish the risk of supply chain attacks. This project was conducted within Rabobank.

Contents

1 Introduction

As society and technology advance, it is required to have larger and more complex applications to assist them in their growth. Open source libraries are crucial in this development, since they allow a developer to reuse the existing code instead of writing everything from scratch which drastically reduces their workload. According to Snyk [30], nowadays 96% of applications use third-party dependencies, where approximately 80% of the application's code comes from open source dependencies. This is the key for building complex applications that accompany the fast-changing needs of the society. However, the issue with the increased use of third-party dependencies is that they can contain malware or security vulnerabilities that attackers may exploit. To confront it, there are available software composition analysis (SCA) tools that help to identify those known vulnerabilities and usually offer a solution, for example, a newer version. Unfortunately, these tools are not perfect and they do not detect all types of security issues. [29] [37]

At Rabobank, developers use Nexus IQ as their SCA solution, which is the motivation behind choosing it to conduct tests within this thesis.
That is precisely the aim of this paper, to research how to detect:

RQ_1: What are the possible attack vectors in supply-chain attacks?

RQ_2: Which markers could be used to detect each category?

RQ_3: For which attack vectors does Nexus IQ provide an alert?

RQ_4: Does the new tool detect attack vectors that Nexus IQ does not?

Firstly, chapter 2 will provide a brief overview of certain technical terms used throughout the paper. It is followed by chapter 3, which will give an introduction to supply chain attacks and illustrate their impact through two examples. Furthermore, it will cover the importance of having an SBOM and SCA tool, together with an in-depth introduction to Nexus. Chapter 4 will define seven possible supply chain attack categories (RQ_1) together with their markers (RQ_2); specific thresholds or characteristics that are used to detect each attack category. In Chapter 5 we will collect and analyze 1000 most popular Python packages, to find the ones that belong to each of the attack categories. Then, we will create a toy application that will use those potentially dangerous PyPi packages, and test it with Nexus IQ in order to investigate whether it would find and report any issues (RQ_3). Chapter 6 will focus on a similar investigation as Chapter 5, only this time, Maven packages will be used. Therefore, we will create a toy application that uses "dangerous" Maven dependencies from each of the seven attack categories, and test it against Nexus IQ to investigate whether it would provide any warnings (RQ_3). Finally, in Chapter 7, we will create our own tool that will be capable of flagging all of the attack categories that Nexus IQ could not. Moreover, it will also provide a health score for open source packages.

2 Background

This chapter gives a brief overview of several technical terms used throughout the paper.

- **Software Development Life Cycle (SDLC)** - is a process that enables the creation of high-quality, low-price software with high development speed. It outlines a detailed plan of deliverables for each SDLC phase, such as planning, development, testing, etc.
- **Software composition analysis (SCA)** - tools automate the process of identifying, analyzing and managing open-source dependencies used in one's application. Specifically, they assess vulnerabilities and verify licensing of open source dependencies. An example is Nexus IQ which will be researched in more detail throughout the paper.
- **Common Vulnerabilities and Exposures (CVE)**[1] - identifies, defines and catalogs all publicly known vulnerabilities and exposures, maintained by MITRE.
- **National Vulnerability Database (NVD)**[2] - is a database which contains all known reported vulnerabilities, which are assigned CVEs. It is run by NIST and sponsored by U.S. Government Homeland Security Department.
- **Package manager** - is a collection of tools that automate installation and the update process of one's dependencies. Some examples are pip, Conda, Yarn, etc.
- **Repository** - contains collections of software that developers may include in their code as a dependency. Repository can be public, accessible to everyone on the internet, or private, accessible only to certain users. Examples of public ones are Pypi for Python projects, npm for Node.js, RubyGems for Ruby, etc.
- **Malicious vs vulnerable libraries** - Malicious libraries contain malware and are usually created by adversaries with the sole purpose of exploit. Vulnerable libraries contain a security flaw which an attacker might try to exploit. They are quite usual and mostly created by regular developers with good intentions.
- **Software bill of materials (SBOM)** - is a list of all third-party and open source dependencies present in a codebase. It also includes a list of licenses, versions of the dependencies used and their patch status. This allows organisations to quickly identify which dependencies are used in case of a vulnerable one. [16]

[1] https://cve.mitre.org/
[2] https://nvd.nist.gov/

3 Supply-chain attacks

This chapter firstly provides an introduction to supply-chain attacks. Section 3.2 covers the possible impact of dangerous dependencies through two examples, Log4J and Equifax breach, together with consequent remediations which are SBOM (chapter 3.3) and SCA tool Nexus IQ (chapter 3.4.

3.1 Introduction to supply-chain attacks

Repositories such as npm, PyPi, RubyGems are very popular amongst programmers since there they can obtain already developed mature libraries from their peers, which heavily reduces their workload. Unfortunately, this is not completely secure because of supply-chain attacks which date back to at least 2016. The first known attack was by a college student whose malicious package found on PyPi got executed over 45,000 times in the time-span of a few months, where about half the time his code was given administrative rights [26]. From that moment, supply-chain attacks have become a regular occurrence and present a challenge to this day. In 2022, a 742% year-on-year growth was observed on supply chain attacks targeting open source software [50]. Since open source repositories present a crucial role to millions of programmers, they are a perfect ground for attackers to try to infect their machines with malicious code. Especially, since repositories often lack vetting controls and robust security. Moreover, communication with the repository's server does not raise an alert of a firewall or an antivirus, since it is a trusted resource [27]. When a developer installs a malicious package, the attacker's malicious code gets executed which compromises the user's machine. There are endless scenarios of what the compromise could be, such as stealing credit card data, login credentials, identity theft, etc.
There are several techniques attackers use to insert their malicious code. The following chapter, chapter 4 identifies and analyses those techniques.

3.2 The impact of supply-chain attacks

This section provides an insight into how big of an impact a dangerous library can have and why it is of extreme importance to apply the patch quickly.

3.2.1 Log4j

Apache Log4j is a simple to use, open source Java library which collects and manages information about system activity, making it very popular amongst developers and widely integrated into other software packages. It was created and maintained by the Apache Software Foundation (ASF), which is a non-profit corporation that provides support for open source projects. The problem occurred in 2013, when a request by a community member to add JNDI Lookup plugin support was reviewed and committed by the Log4j project team. The addition allowed an attacker to perform arbitrary code execution loaded from LDAP servers, since JNDI features do not protect against attacker controlled LDAP. This brings up the issue and is an example of insufficiently maintained libraries which is covered in the next chapter (4.5). According to the Cyber Safety Review Board [19], the vulnerability could have been caught in 2013 if the review was focused and performed by someone with sufficient experience on JNDI support. However, the same problem arises, there was simply not enough resources to perform such a review, especially since the developers were on volunteer-basis.

In the end, the vulnerability was only noticed and reported to the ASF team on November 24, 2021 by a security engineer from the Alibaba Cloud Security team. Since the flaw impacted virtually every networked organisation, ASF responded quite quickly with an official fix followed by a public disclosure of appropriate CVEs. Exploitation occurred rapidly after the publication, with approximately 400 exploitation attempts per second. In response, thousands of security experts across the globe assembled to identify hundreds of millions of possibly infected devices,

which culminated into one of the most rigorous cybersecurity response in history. However, organisations spent significant resources to deal with it, and experts claim that vulnerable instances of log4j will remain in systems for many years [19].

3.2.2 Equifax data breach

In March 2017, a vulnerability was discovered in an open source framework called Apache Struts which allowed the attacker to send malicious code in the "Content-Type" header of an HTTP request [24]. Equifax, a global data, analytics and technology company used vulnerable Apache Struts dependencies for creating enterprise Java applications. Even though ASF published the vulnerability and released a patch, Equifax failed to apply it due to mistakes in internal processes. Attackers exploited this vulnerability, and hacked the company through a consumer web portal, from which they moved on to other servers and pulled the data out of the network. It resulted in a severe data breach since 143 million people were potentially affected with their names, dates of birth, addresses, social security numbers and driver licence numbers leaked. Moreover, a smaller subset of the records also included leaked credit card numbers, which all added together, could have lead to fraud and identity theft [24].

In the end, the company had to agree to a global settlement which includes up to 425 million dollars, to help the people affected by the data breach. [22]

3.3 SBOM

These two examples, together with many others, show how big of an impact on an organisation, or even on a society in general, a vulnerable dependency can have. Since owners of open source projects do not know who are their users, it is hard to identify and warn them in case of a vulnerability. That is why it is important for organisations to stay on top of major cyber security related news and use appropriate tools to perform scans and identify vulnerable versions (such as Nexus IQ), after which, they should apply the patches rapidly.
However, it can be difficult for organisations to keep track of which dependencies, versions, and licences were used, especially when an application is already in production. According to Sonatype, less than 50% of organisation know which software components make up their applications [49]. Consequently, even when aware of a new major vulnerability, it is very difficult to identify vulnerable applications and apply a fix. That is why companies should create Software Bill of Materials (SBOM) when deploying an application. It is a list that keeps track of all third-party and open source dependencies present in a codebase. SBOM is even listed as one of the requirements for companies in President Biden's *2021 Cybersecurity Executive Order* [17], where he urges enhancement of software supply-chain security. Nexus IQ is one of the SCA tools that automatically creates SBOM; further details are explained in the following chapter.

3.4 Nexus IQ

As already mentioned, developers at Rabobank use Nexus IQ as their software composition analysis (SCA) tool, which is why in this chapter, we will dive deep into it. We will provide an introduction into Sonatype's Nexus IQ, its tools and available features.

3.4.1 Introduction to Nexus

Nexus IQ is Sonatype's policy engine server that is powered by precise threat intelligence on open source dependencies. It creates the "intel" data from publicly available information, using automated vulnerability detection system which monitors, correlates, aggregates and incorporates machine learning. Sonatype collects the data from many different sources, such as NVD, email lists, website security advisories, GitHub events, blogs, OWASP, OSS Index, customer reports, Twitter, etc. Moreover, in the combination with automated identification, it also uses human

research which is supposed to eliminate false positives and negatives [46]. Nexus IQ consists of several different tools, which are briefly described in the following text [45]. It is also important to mention that Rabobank, at this moment, uses only Lifecycle and Repository as their Nexus IQ solution.

- **Nexus Lifecycle** - SCA tool that automatically finds and fixes known open source vulnerabilities at every stage of SDLC. More precisely, it can integrate with various IDEs and deliver precise component intelligence in the code writing phase. In the build and deployment phase, it provides automated policy enforcement, and after the application is in the production, it continues to monitor for vulnerable components. When an issue is spotted, Nexus Lifecycle provides an alert together with the possible fix, or it even fixes the issue automatically depending on organisations policies. Furthermore, it creates SBOM which allows the user to gain visibility into all of the used dependencies, together with the transitive ones [42].

- **Nexus Auditor** - SCA tool that scans third party applications for any restricted licences or known security vulnerabilities, which aids the developer in his choice. It continuously monitors production applications to identify newly disclosed vulnerabilities. Moreover, it automatically generates SBOM to identify open source dependencies [39]. The main difference from Lifecycle is that Auditor is aimed at monolithic applications that have little to none ongoing development. So, it is a better choice for legacy applications which are integral to the organisation and need to meet various industry standards. Whereas, Lifecycle is an optimal choice for applications that undergo active development and upgrade process, since it enables various integrations and scanning throughout the entire SDLC [28].

- **Nexus Firewall** - It is an early warning detection system which prevents malicious open source components from being downloaded into organisation's repository. Specifically, it is powered by Nexus Intelligence and aims to secure SDLC by automatically blocking malicious and suspicious components from entering. Furthermore, if possible, it automatically returns secure versions of the component in question. The firewall pipeline starts with a new component trying to be included into SDLC. Before entering, the component is evaluated through AI behavioral analysis and automated policy enforcement, which can result in three labels. Either the dependency is known to be malicious and stays in quarantine or it is known to be safe and enters organisation's pipeline. The third option is that it is found suspicious, in which case it is quarantined until research team reviews the component. Then, it is either found safe and automatically released into organisation's pipeline, or malicious and therefore blocked [41].

- **Nexus Repository** - An artifact repository manager that teams can use to store their artifacts and share it amongst each other. It is also used to store third-party open source software with native package manager compatibility. It supports various ecosystems such as Maven, npm, PyPI, RubyGems,etc., and distributes containerized apps like Docker. It is compatible with various IDEs and CI such as Eclipse, Visual Studio, Jenkins, IntelliJ, etc. When integrated with Nexus Firewall, the repository should consist of "safe" dependencies since it proactively blocks malicious ones. Moreover, it supports private hosted repositories and enables role-based access controls. It works as a proxy because it fetches dependencies from public registries in case they are missing. Another useful feature is repository health check which provides guidance on what should be upgraded or revoked, lists the number of times components were downloaded and ranks them. Furthermore, it provides monthly reports on the health of the code pipeline. [43]

- **Nexus Container** - Assists teams to discover, continuously monitor and fix container flaws throughout the entire lifecycle. It identifies vulnerabilities during code development

by monitoring images in registries and running automated tests for security compliance. It removes the need for manual compliance enforcement since it uses auto-learning combined with behavior analysis to build the security policies automatically. Sonatype company claims that it is the only solution that can enforce Data Loss Protection and tackle zero-day malware, tunneling and breaches [40].

4 Attack vectors

This chapter defines seven categories of possible attack vectors together with their markers; used to identify each category, where some of them are collected from research (academic papers, articles, blogs) and some were identified by us. Certain markers have higher accuracy, while others have to be combined with multiple ones and put in contexts to provide higher assurance of a potential attack vector. That is why, they are sorted in a descending order, i.e. first ones have higher accuracy while the last ones have a lower one. Furthermore, it is important to highlight that the attack categories differ in properties, i.e. certain categories, such as typosquatting, dependency confusion and hostile takeover are examples of a *direct* attack. On the other hand, abandoned, insufficiently maintained and vulnerable dependencies are examples of an *indirect* attack, i.e. adversaries can misuse them as an entry point to deploy an attack. Transitive ones are a category on its own, since they can depend on each of these dangerous dependencies (overlap with previous six). Furthermore, it is important to mention that there is an overlap between abandonware (4.4) and insufficiently maintained libraries (4.5), since abandonware is an extreme case of insufficiently maintained ones. Therefore, certain markers between these categories will also overlap.

4.1 Typosquatting

This attack relies on the user making a typing mistake. Essentially, the malicious package containing malware has a name that slightly differs from the name of the legitimate package, usually one letter difference (e.g. pillow instead of pillow, djangoo instead of django). In haste of the moment, the programmer may not notice that he/she is installing the wrong package, the malicious one. To make things worse, it is challenging to notice the mistake even afterwards, because the attacker's package usually includes the same code and functionality as the legitimate one that is being impersonated, plus the concealed malicious code. There are two additional approaches that attackers can use; called brandjacking and combosquatting. In brandjacking attack an adversary creates an impression that the package is coming from a legitimate source, such as AWS-package_name. On the other hand, in combosquatting attack an adversary adds a prefix or postfix on a legitimate package name. To give an example, *discord-lofy* and *discord-selfbot-v14* are two malicious packages that steal Discord tokens, and are an impersonation of a popular legitimate *discord.js* package [27]. Therefore, programmers should be extra cautious and double-check the spelling and library in question before the installation.

Markers:

T_1: Similar name as a popular legitimate package - usually the name differs in one to few characters, or it has an extension [27]

T_2: Not installed often - since the attack relies on the developer making a mistake, typosquatted packages are usually not installed often, or at least less then the legitimate popular package that they are trying to impersonate

T_3: Same/Similar purpose as the legitimate package

T_4: Contains malware

T_5: No previous versions/new package

4.2 Dependency confusion

Dependency confusion is a relatively new type of an attack which relies on automated process of installing and updating dependencies with package managers, and was discovered by a researcher

Alex Birsan [18]. The attack vector are malicious dependencies published on a public repository with the same name as internal dependencies from a certain organisation. The problem occurs when the package manager used by that organisation checks the public repository either before or in addition to the internal repository. Then, the malicious dependency gets used rather than the trusted one. Moreover, hackers can even give a higher version number to their dependency which would make their application automatically use it. According to Birsan, it is possible to find these dependency names even though they are stored in private repositories. Some names he found on GitHub, on various internet forums, through leaked internal paths or *require()* calls. But, by far the best place was inside javascript files since it is quite common for internal *package.json* files to be embedded into public script files through their build process. In his research, he successfully attacked over 30 big, well known companies, such as Microsoft, Amazon, Tesla, Apple, etc. Only few hours after publishing his work, the attack became popular resulting in many copycats [18] [53]. Figure 1 shows the timeline of the attacker reaction time.

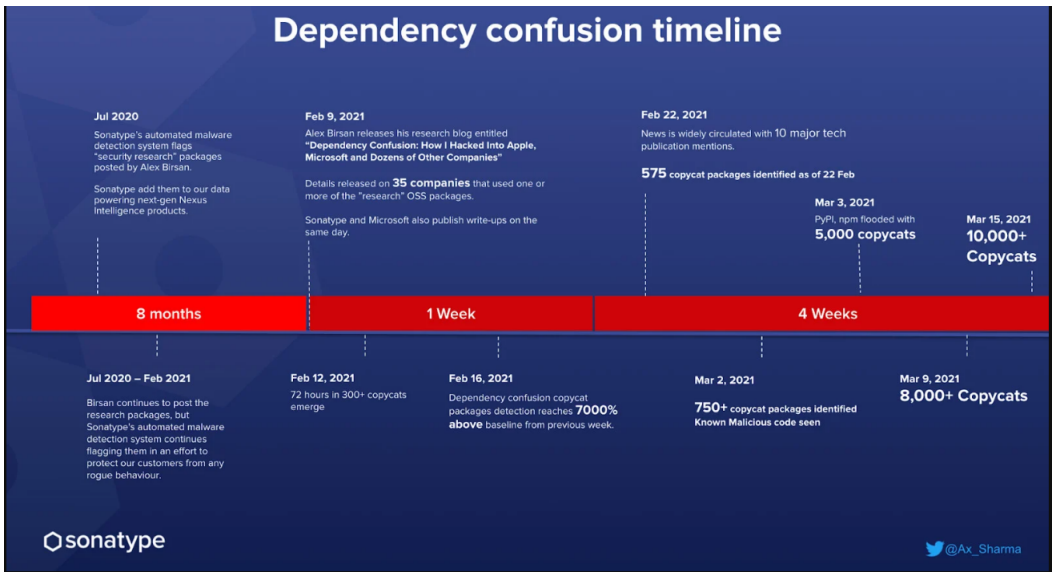

Figure 1: Attacker reaction timeline, after the attack was public

We can see that in a matter of days from publishing the findings, the copycat level reached 7000% increase. Therefore, to protect the organisation, one should use a package hosting service that has a reliable workaround against dependency confusion attack. However, another alternative is to simply reserve the package name on the public registry.

Markers:

C_1: Same name as the internal dependency

C_2: Higher version

C_3: Contains malware

4.3 Vulnerable dependencies

A vulnerability is a flaw or a weakness in a software that could lead to an exploitation by an attacker. These flaws are usually not intentional, especially since any programmer can publish his library to the public repository, mistakes can be made. For example, forgetting to validate user's input could lead to an attacker performing a buffer overflow. According to the 2022 OSSRA report [52], 2,409 codebases were analysed and 78% contained open source code, where 81%

contained at least one known vulnerability. Luckily, CVE and NVD define and catalog all publicly known cybersecurity vulnerabilities, which helps developers to identify them in the libraries they use. Moreover, SCA tools such as Nexus automate that process since they check the code and used dependencies against those databases. However, it is still possible that there are some unknown or undetected vulnerabilities as well as new ones arising after some time, which is why it is important that the libraries are in constant evolution. The community, i.e. developers using those libraries, present a crucial role since they can report any found vulnerabilities and help to patch them. That is why libraries should continuously go through changes and upgrade their versions. Therefore, it is very important to use the latest version of the library because that is the one with the most recent security patches, even when the switch presents more workload. Also, when possible, it is usually better to choose a library with a big community because it means there are more people to report an issue [59].

Markers:

V_1: Contains at least one CVE - publicly disclosed cybersecurity vulnerabilities

V_2: Contains at least one vulnerability that can be found in NVD - database maintained by NIST

V_3: Contains a vulnerability that is assigned a CVSS score greater than 1 - measures exploitability, impact and scope

4.4 Abandonware

Abandoned libraries present an issue because it means that they are outdated and that there is nobody maintaining them, i.e. patching the security flaws and producing newer versions. Unfortunately, they are a very usual occurrence since the developers often grow bored and abandon their projects. Usually, SCA tools do not trigger a warning since there is no newer version available, so abandoned libraries are oftenly used in code. According to the 2022 OSSRA report, from 2,409 tested codebases, 88% contained components that had no new development in two years and 85% contained open source that was more than four years out of date [52]. Therefore, abandoned libraries present big security issues and we will look into how to detect them in the following chapters.

Markers:

A_1: No new release in more than two years [55]

A_2: The project is archived (read-only state) on the project's home page (e.g. GitHub) [20]

A_3: No activity from the owner on the project's home page (or no home page found) [55]

A_4: None of the issues or PRs got closed in the past year[56]

A_5: Annotated as abandoned/deprecated either in the terminal when user installs it or in the *README* [34]

A_6: Depending on non upgraded dependency versions

A_7: Provides a redirect to a different package [6]

A_8: Community raising questions whether the project is abandoned (e.g. in the issue section of GitHub)

A_9: Owner is unreachable [55]

4.5 Insufficiently maintained libraries

When an open source software is maintained by a few people or even only by a single person things can go horribly wrong. Especially, when those people are maintaining the library for free, on a volunteer basis. This means relying on a few or even a single developer not to make a mistake and to make fast updates. Similarly as abandonware, these libraries present a security issue and are an often occurrence on public repositories. Even if they do not have vulnerabilities when they are published, it is a high chance that they will have later on because of improper maintenance. A recent example of usage of an insufficiently maintained library gone wrong is Log4j, which had a significant impact on organisations. More about that in the Impact section [58]. It is important to highlight that the abandonware is an extreme case of insuficientlly maintained libraries, which is the reason certain markers overlap.

Markers:

I_1: No new release in more than a year [56]

I_2: No new commits in a year on the projects homepage (e.g. GitHub)

I_3: Less than half of the issues or PRs closed in the past year [56]

I_4: Depending on non upgraded dependency versions

I_5: One maintainer

I_6: Annotated as not properly maintained in *README*

I_7: Community raising issues about project's maintenance (e.g. in the issue section of GitHub)

I_8: Project is looking for funds/assistance:
- npm repository: *npm fund* [< *pkg* >] - this command retrieves information on whether the project needs funding and it tries to open it's funding url [35]

I_9: Maintainer/s slow to close an operation such as an issue or change request - according to CHAOSS metrics [9] the average time to close an operation for open-source projects is 8.941 days. We argue that the acceptable time-frame should heavily depend on the type, severity and complexity of the request (e.g. vulnerability patch vs feature request). Therefore, the maintainer is slow to close an operation when there is a high number of issues that stay open for a long period of time.

I_{10}: Low number of contibutors (every distinct email address that has a commit merged into the project) - higher number of contributors over last 12 months results in higher chance that the project is receiving proper updates, support and resources [9]

I_{11}: Maintainer-to-contributors ratio - there is a higher chance for a package to contain security issues when a single maintainer is responsible for many contributors. Moreover, it is easier for an attacker to bypass maintainer's radar and insert malicious code. [60]

I_{12}: Overloaded maintainers - when a maintainer owns a large number of projects, it can negatively affect maintenance [21]

I_{13}: Personal accounts vs official company accounts - company accounts are likely to be more trustworthy and provide proper maintenance

4.6 Hostile takeover

Hostile takeover is an attack that can even compromise "safe" libraries; the ones that do not contain vulnerabilities or malware, and are properly maintained. This means that hackers usually target accounts of maintainers of popular projects, that millions of developers use and trust. When a malicious actor gains access to the maintainers account, he can publish malicious versions of the project that can perform various malicious tasks, such as installing a cryptominer, stealing credentials, tokens, etc. Research into this attack by Nikita Skovoroda [38], resulted in obtaining direct publish access into 14% of npm packages, including the popular ones, and an estimated 54% of packages that were potentially reachable through dependency chains. The method used was a combination of brute-force attack, known account leaks and npm credential leaks mostly from GitHub. Fortunately, it was a research instead of a malicious act, so the outcome was a positive one, a mass password revocation by npm. However, it should serve as a warning because it shows how easy it can be for hackers to obtain access to projects. With that being said, there are many other examples where attackers did hijack projects, such as the *coa* and *rc* library takeover, where the malicious script contained a password stealing trojan. Both libraries are extremely popular with few millions of weekly downloads and at that time, no new release in 3 years, which could give a clue how attackers choose their targets [15]. Another way an attacker can insert malware into a popular package is using social engineer techniques to manipulate current maintainers into adding him as a maintainer. Then, he can publish malicious versions of the package and even remove the original maintainer to become the sole owner. A variant of this attack is when the attacker manages to inject malicious code through pull requests, via compromised development tools, or commits when he is given commit rights on the repository of the project [61]. Therefore, this attack should not be taken lightly, so some companies scan the repositories with malware detection system and take into account reports made by community. One of them is Sonatype, whose Nexus Intelligence automatically detects malware, and every package that gets flagged as suspicious undergoes further inspection while Nexus Firewall protects its users from installing these suspicious packages [15].

Markers:

H_1: Sudden ownership transfer - previously abandoned project suddenly has a new owner. An attacker can trick the project owner to transfer him the ownership by showing interest and enthusiasm in maintaining a popular abandoned project [60]. It is not a strong marker since a legitimate developer can also take over the ownership, however, it should raise caution. One should look at the previous activity of the new owner, how much information about him is available, whether he has other projects or reports to get a clearer context.

H_2: Account takeover - project's security heavily depends on the security of it's maintainers accounts. If an attacker steals the credentials, he can publish malicious versions of the project under maintainer's name. This attack is quite challenging to notice, so one should look out for any statements about the breach from the maintainer on social media and any reports made by the community. [61]

H_3: New version contains malware/contains an installation script - an attacker can misuse an installation script to perform malicious commands that, for example, steal sensitive data or create a backdoor access. The presence of an installation script should not automatically mark the package as malicious, but it should raise suspicion. According to Zahan et al. [60], 93.9% (3,412) of malicious packages contained at least one install script, which indicates it is a frequent form of an attack. Attackers use this approach while hijacking a project to hide the presence of the malicious code.

H_4: Expired maintainer domain - an attacker can search for popular unmaintained packages that have an expired domain. Then, he can check the domain availability in a domain registrar

(e.g. GoDaddy), purchase it and alter the MX record in order to steal maintainer's email address. Moreover, if the maintainer uses the same email address in the package repository (e.g. npm) without a 2FA authentication, an attacker could reset the npm account and hijack the project. [60]

H_5: Inactive maintainers - there is a higher chance that attackers will target packages with inactive maintainer(s) since the attack will remain undetected for a longer period of time [60].

H_6: Large number of maintainers - when a project has too many maintainers it is more susceptible to be hijacked. It provides an attacker with more targets to try out social engineering attacks on and consequently, perform account takeovers.

4.7 Transitive dependencies

Libraries can depend on other libraries that can also depend on other libraries which can go on for quite some layers. The issue with the inheritance is that the "parent" library mostly inherits security flaws from the "child" one which makes the bugs flow around quite rapidly and makes them hard to identify. That is why it is not enough to only check if the direct dependencies are free of security flaws. It is almost impossible for developers to go through that "dependency tree" manually because it would take a lot of effort and it can become very complex quite quickly. However, some of the SCA tools do provide support and are able to identify vulnerable transitive dependencies. For example, Nexus provides a graphical view of the dependency tree, which makes it easier to identify and trace vulnerable dependencies even when deeply nested [25].

Markers:

T_1: Dependency depth of at least 2 layers

T_2: At least one of the indirect dependencies belongs to one of the six categorizes from above

5 Python Application Test Using Nexus IQ and PyPi

In this chapter, we will create seven toy applications that use potentially dangerous third-party dependencies from one of the seven attack-vector categories, and test them with Nexus IQ in order to investigate whether it would find and report any issues to answer **RQ3**(*"For which attack vectors does Nexus IQ provide an alert?"*). The reasoning behind why and how certain dependencies were chosen is explained in sections 5.1, 5.2 and the paragraph *"Library selection"*, which can be found at the beginning of each attack-vector analysis sub-chapter. The results of Nexus IQ test can be found in the *"Results"* paragraph which is the final paragraph of each attack-vector analysis sub-chapter. Also, it is important to mention that every time we refer to Nexus IQ, we mean Nexus Lifecycle and Nexus Repository since those are the solutions Rabobank uses.

5.1 Pypi Data Collection

For the purpose of data collection, we built a web scraper to help us later choose appropriate libraries for toy application creation. It scraped through first 1000 most popular packages on Pypi (*file obtained from [57]*); with popularity determined based on the number of downloads in the past month (*September 2022*). We only chose time-span of the last month in order to keep timely statistics on popularity of packages, i.e. we were not interested in packages that were very popular years ago, but not at the present moment. Therefore, using the web scraper, we collected package names, number of downloads (in the last month), last release date, name of the author, name(s) of the maintainer(s), project's home page, number of releases and finally, development status. An example of scraped data for one package can be seen in Figure 2.


```
project_name: "python-dateutil"
download_count: 166700747
release_date: "Jul 14, 2021"
author: "Gustavo Niemeyer"
development_status: "Development Status :: 5 - Production/Stable"
home_page: "https://github.com/dateutil/dateutil"
number_of_past_releases: 32
▼ maintainers: [] 4 items
  0: "jarondl"
  1: "dateutilbot"
  2: "pganssle"
  3: "tpievila"
```


Figure 2: Scraped data of "python-dateutil" project

We hypothesized that by analysing this data, we would be able to establish which libraries align with certain markers mentioned in sections 3.2.1 - 3.2.7. For example, only one maintainer would align with I_4, while many maintainers with H_6, last release date older than a year with A_1, authors of multiple projects with I_{11}, etc. After identifying the subset of potentially dangerous libraries, we would manually check each of their project's homepage to inspect if they still exist or to obtain further information about the project. Moreover, if possible, we would manually examine the number of contributors, open issues, open pull requests, any community concerns, sudden ownership transfers, etc.

5.2 Pypi Data Analysis

This chapter analyzes the collected data from section 5.1 in order to guide our *"Library selection"* in the following chapters. It groups the data on certain interesting features, which are inspired

by the markers from chapter 4.

Figure 3 shows how many packages (out of the top 1000) did not have a new release since n years ago.

Latest release older than:	One year ago *(2021)*	Two years ago *(2020)*	Three years ago *(2019)*	Four years ago *(2018)*	Five years ago *(2017)*	> six years ago *(2016-2010)*
Number of packages:	129	59	34	24	13	24

Figure 3: Relation of the latest release date and the number of packages

It is interesting to see that amongst 1000 most popular projects on Pypi, with millions of daily downloads, there was a lot of them that do not have an up-to-date latest release (according to previous markers). To be more specific, 154 projects did not have an updated version for more than two years.

Figure 4 shows five of the oldest identified projects (no new release in "N" years) in descending order, together with their monthly download count for September 2022.

Package name:	*"docopt"*	*"pycrypto"*	*"pathtools"*	*"sgmllib3k"*	*"crcmod"*
Latest release date:	*16.06.2014.*	*17.10.2013.*	*25.08.2011.*	*24.08.2010.*	*27.06.2010.*
Download count (September 2022):	*11,164,325*	*5,374,143*	*2,345,868*	*3,017,288*	*5,594,094*

Figure 4: The oldest identified packages

It is quite alarming to see that projects with several million monthly downloads did not have a new version in approximately 10 years; with the oldest *"crcmod"* not having a new version for 12 years.

Figure 5 shows how many projects have n number of maintainers.

Number of maintainers:	*Zero maintainers*	*One maintainer*	*Two maintainers*	*Three maintainers*	*>= Four maintainers*
Number of packages:	*0*	*358*	*273*	*219*	*186*

Figure 5: Relation of the number of maintainers and the number of packages

It is quite positive to see that there are zero packages that have no maintainers; which would mean the author is the only maintainer. However, it is important to mention that it is not completely accurate since authors can also list themselves as maintainers. In that case, our applied analysis would still count as if there is one maintainer, since the name in the "author" field differs from the username used in "maintainers" field. One of the found examples is *"jaraco.classes"* project with the author *"Jason R. Coombs"* and maintainer *"jaraco"*. After manual investigation, we concluded it is indeed the same person, therefore, this is one of the examples which should be placed in the zero maintainer column. On the other hand, 358 is quite a high number of packages to have only one maintainer. Especially, considering we are observing top 1000 most popular packages at the moment, it is hard to imagine that one maintainer can properly deal with all the patches, updates, issues, pull requests, feature requests, etc. We have to mention that 358 is not a completely accurate number, since some projects, such as *"boto3"*, have a team of people behind one maintainer account, such as *"aws"* (usually a company account). It is also

possible that maintainers are mentioned in the author field together with the author instead of the maintainer field (e.g. *"pytest"*). Furthermore, the project with the highest number of maintainers was *"zope-interface"*, with 30 maintainers.

Figure 6 shows how many authors own n number out of the 1000 most popular packages.

Number of packages:	*Zero packages (error)*	*One package*	*Two packages*	*Three packages*	*>= Four packages*
Number of authors:	*97*	*497*	*48*	15	*21*

Figure 6: Relation of the number of authors and the number of packages they own

The first column "Zero packages" is marked as error since it does not indicate that 97 authors own zero packages. Instead, it implies that for 97 projects it was not possible to scrape the name of the author since it was not filled in on the Pypi website (e.g. *"frozenlist"*). The rest of the analysis results was as expected, i.e. the majority of the authors own one package, which is logical since it is quite hard to create, and then properly handle more than one of the most popular packages out there. The last column indicates that 21 author owns four our more packages, which we found quite interesting and decided to look into in more detail. That is why, *Figure 7* shows seven authors with the most packages out of the 1000, in descending order.

Authors:	*"Microsoft Corporation"*	*"Google LLC"*	*"Google Inc."*	*"Amazon Web Services"*	*"Jason R. Coombs"*	*"Sébastien Eustace"*	*"Jupyter Development Team"*
Number of packages:	*111*	*28*	*14*	12	*11*	*11*	*10*

Figure 7: Top seven authors (own the maximum number of projects)

The results align with our hypothesis; it is challenging for a volunteer developer to create and handle more than one popular project from our subset. That is why, it is not surprising that five out of top seven authors are companies, with the maximum of 111 packages owned by "Microsoft Corporation". On the other hand, the results for a single person author were quite interesting and surprising. Eleven is a very high number of packages to own, especially out of the 1000 most popular ones. This discovery should straight away "trigger" many of the markers previously explained, which is why we choose these two authors and their projects to be examined more deeply, especially within the hostile takeover and insufficiently maintained library attacks.

Figure 8 shows how many packages have a certain development status.

Development status:	*Pre-alpha*	*Alpha*	*Beta*	*Production/Stable*	*Mature*	*Inactive*	*Not declared*
Number of packages:	*3*	*35*	*162*	563	*25*	*7*	*205*

Figure 8: Number of packages in respect to the development status

The high number of packages found in the pre-alpha, alpha and beta stage was quite surprising. This could indicate that 20% of the most popular packages are in experimental version, which is not ready for a public release. Furthermore, it could indicate that the software was not properly tested and that the application is not stable, some features might not work properly, or

that the author released the first version and did not follow through with the upgrades. That is why we will look into these packages in more detail within the insufficiently maintained libraries markers. On the other hand, a dozen of mature packages and the high amount of stable packages was expected to be identified within the scope of the most popular packages. Seven packages were classified as inactive, which we will look deeper into within the abandonware analysis (5.3). Finally, 205 packages did not have a specified development status.

5.3 Abandonware Nexus IQ Test

In this section, we will choose which abandoned libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
We decided to firstly investigate packages found in Figure 4, since they are an extreme case of the latest release date and, therefore, a good indication that the projects are indeed abandoned. Secondly, we looked into the development status assigned to those packages and interestingly, none of them had the inactive status. This is why, on top of these packages, we decided to also choose one with the inactive status *(Figure 8)*. We chose *"oauth2client"* since it, nevertheless, had a very high monthly download count; 19,105,877.
Finally, we manually examined other relevant information for those three packages on their home pages, to find which markers apply. The list below shows the results:

- **crcmod** - There was no new release in 12 years, the project was not archived on Github, but there was no new activity in 12 years and the issues were not closed. The project also has an official homepage, which seems as it was not updated in a while (e.g. copyright year is 2010). Moreover, it does not have any dependencies, so it did not depend on non upgraded ones. Therefore, applicable markers are: $\boldsymbol{A_1, A_3, A_4}$.
- **sgmllib3k** - No new releases in 12 years, homepage did not exist anymore and it did not have any dependencies. Furthermore, the author annotated in Pypi that he does not intend to maintain the package and there were some concerns raised by the community[7].
  Therefore, applicable markers are: $\boldsymbol{A_1, A_3, A_5, A_8}$.
- **oauth2client** - No new releases in 4 years, it was clearly stated both on Pypi and projects home page that it is deprecated, could depend on non-upgraded packages (requires higher or equal version).
  Therefore, applicable markers are: $\boldsymbol{A_1, A_2, A_3, A_4, A_5, A_6, A_8}$.

The last package was already confirmed to be abandoned (inactive development status), so the fact that a lot of our markers apply further proves that the choice was good. It is also important to highlight that we did not check if the owner is reachable (A_9). Figure 9 illustrates an example of how authors mark their projects as deprecated.

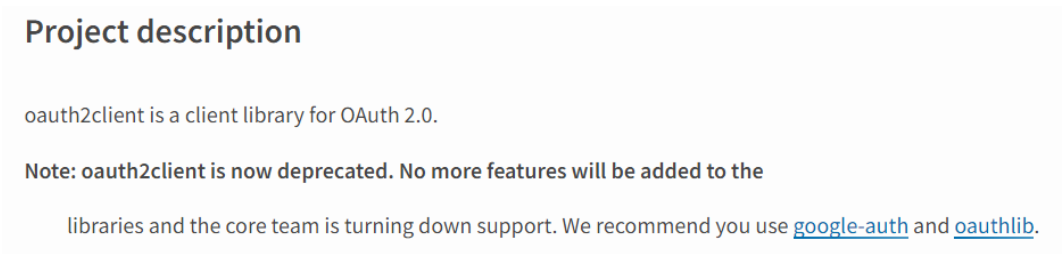

Figure 9: Deprecation notice for oauth2client project

Hypothesis
Our conclusion is that the three libraries from above are abandoned and we are interested whether

Nexus IQ would detect it in order to answer **RQ3**(*"For which attack vectors does Nexus IQ provide an alert?"*. We hypothesized that Nexus would detect certain vulnerabilities, since the packages are not updated for a long time and could depend on non-upgraded dependency versions, but we were not certain if it would warn us that the packages were indeed abandoned.

Results
Firstly, we only imported the libraries in the toy application, without using their code. The results were quite surprising, Nexus IQ did not find anything to report on. Then, we specified a non-upgraded dependency version *httplib2 0.17.0* of *oauth2client* in *requirements.txt* that had a known CVE. In this case, Nexus IQ found that the *httplib2 0.17.0* is vulnerable and contains a CVE (as we hypothesized), but it again did not report anything about the libraries actually being abandoned.

Finally, we added a function from each library to the toy application to see if the results of Nexus IQ report would change when the libraries are actually used; the results were unchanged. Therefore, our conclusion is that Nexus IQ is not capable of detecting abandoned libraries, even the ones marked as deprecated by the author and in read-only state.

5.4 Insufficiently maintained libraries Nexus IQ Test

In this section, we will choose which insufficiently maintained libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
To identify insufficiently maintained libraries, we used an intersection between collected data from *Figure 3, Figure 5 and Figure 7*. That means that we looked only at libraries with latest release date older than a year, with one maintainer and with an author that owns more than one library out of the 1000. We used this approach because it aligns with previously stated markers, and therefore, gives a good indication that the libraries found are indeed insufficiently maintained. We decided to concentrate on projects owned by *Jason R. Coombs* and *Sébastien Eustace* because they were authors with personal accounts that owned the most projects. Microsoft, Google and Amazon were behind accounts that owned more projects, but since those were company accounts, we concluded that their projects should be properly maintained. After manually inspecting projects from the two authors, we concluded that Coombs's projects are sufficiently maintained (did not align with the markers) and identified "cachy" as a possibly unmaintained one from Eustace. The list below shows chosen projects:

- **cachy** - No new release in 3 years, no new commits in a year, no PRs or issues closed in the past year, depends on non upgraded dependency version *("redis" >=3.3.6,<4.0.0)*, no contributors.
  Therefore, applicable markers are: $\boldsymbol{I_1, I_2, I_3, I_4, I_5, I_9, I_{10}, I_{12}, I_{13}}$
- **flask-babel** - No new release in 2 years, two commits in the last year, less than half PRs and issues closed in the past year, 30 contributors, 2 maintainers, the latest documentation website is not reachable on Pypi (fixed on GitHub).
  Therefore, applicable markers are: $\boldsymbol{I_1, I_3, I_9, I_{11}, I_{12}, I_{13}}$
- **toml** - No new release in 2 years, less than half PRs and issues closed in the past year, 49 contributors, the author is the only maintainer, one commit in the past year. Therefore, applicable markers are: $\boldsymbol{I_1, I_3, I_5, I_9 I_{11}, I_{13}}$

It is important to highlight that it can be quite challenging to unambiguously determine if the library is insufficiently maintained. If we compare it to e.g. abandoned libraries, the markers for insufficiently maintained ones are much more vague, so it can be harder to determine e.g.

what is a low or high number of committers. That is why our chosen examples align with most of the markers, to be certain that these examples are indeed insufficiently maintained and leave no space for ambiguity. Furthermore, we chose "cachy" as the first example because it had very strong markers, which made it almost fall within the abandonware category.

Hypothesis
Our conclusion is that libraries mentioned above are insufficiently maintained and we are interested whether Nexus IQ would detect it in order to answer **RQ3**. Based on the abandonware results (5.3), we hypothesized that Nexus would not warn us that the libraries are not sufficiently maintained.

Results
Similarly to the abandonware test (5.3), we examined the toy application firstly with only imported libraries in question, and then again after adding some functions from each library. The results were the same, Nexus IQ did not provide any warnings. Therefore, our conclusion aligns with the initial hypothesis, Nexus IQ is not capable of detecting insufficiently maintained libraries.

5.5 Hostile takeover Nexus IQ Test

In this section, we will choose which libraries to use in our toy application. The selection will be split into two categories; libraries that are susceptible to hostile takeover and already taken over ones. Then, we will scan the application with Nexus IQ and examine the results.

Library selection
We decided to split the library selection into two categories; libraries that are more likely to be taken over, and libraries that had an actual hostile takeover.

According to the markers H_1 and H_5, libraries that have a higher chance of being taken over are very popular ones since it creates a bigger attack surface, and abandoned ones since the takeover remains unnoticed for a longer time period. For this example, we decided to choose libraries from chapter 5.3 *(crcmod, sgmllib3k, oauth2client)*, since their properties align with the previously mentioned markers, i.e. they are amongst 1000 most popular projects on Pypi and are abandoned. Furthermore, we decided to also look at libraries from chapter 5.4 *(cachy, flask-babel, toml)* since they are not maintained often, and this attack could also stay under the radar for a longer period of time. Especially *cachy* and other projects owned by Eustace, since his account is a personal one and he owns 11 projects out of the 1000 most popular ones. This makes such accounts more appealing to adversaries because the attack surface is automatically bigger. For example, by taking over Eustace's account, an attacker could automatically endanger 11 popular projects instead of just one. We also identified projects that are searching for new maintainers to transfer the ownership to. One of them is *pytest-forked* with $5,371,968$ downloads only in September, 75 dependent packages and 468 dependent repositories which makes it quite tempting to the attackers [33]. *Figure 10* illustrates an example of potential hostile takeover through ownership transfer.

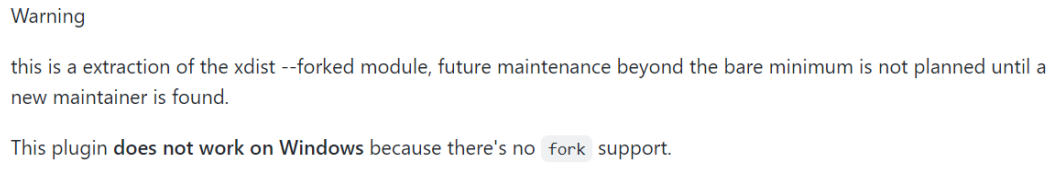

Figure 10: Potential ownership transfer of pytest-forked [4]

Finally, we investigated marker H_4, to identify whether certain libraries are susceptible to

expired maintainer domain attack. We did not check for all 1000 projects, but only for the ones mentioned above (abandoned and insufficiently maintained examples), and for the ones marked as inactive in the development status. The findings showed that the homepage of *"crcmod" (*$5,594,094$ *downloads) and "avro-python3" (*$7,401,097$ *downloads)* was available for purchase. Moreover, *"avro" (*$3,870,354$ *downloads)* shares the same domain homepage as *"avro-python3"*, which also makes it susceptible to hostile takeover. This means that an attacker can, after obtaining the domain, gain control of the author's account and subsequently, hijack the project. This would have a tremendous impact, since the attack surface is quite vast, i.e. both libraries have an immense monthly download count. However, quite recently critical projects on Pypi, which are the top 1% ones based on download count over past 6 months, require 2FA authentication to be able to make any changes [5]. Based on this, we concluded that the top 1000 projects (including *"crcmod, avro-python, avro")* should not be susceptible to this attack instance, since stealing the maintainer's email address would not be sufficient to make any changes on Pypi.

For the second category, we searched for libraries that were, at that moment, under a hostile takeover. It was not possible to find such libraries since Pypi's team actively reviews reports of new malicious releases, takes them down and restores maintainer's account [23]. That is why we decided to slightly shift the investigation and look into known previous examples and compare them with our markers to see if they align. It is important to highlight that we did not check for marker H_3, i.e. the presence of an installation script.

- **CTX** - This library was hijacked in May 2022, by exploiting an expired domain. The project did not have any updates in 8 years, when it suddenly had a new malicious version on Pypi that was used to steal environmental variables (GitHub repo did not reflect any changes). After the community started raising concerns, Pypi team investigated the issue and took down the library [54].
  In conclusion, our markers align, and the applicable ones are: $\boldsymbol{H_2, H_4, H_5, H_7}$

- **exotel** - This library was hijacked in August 2022, through a phishing campaign that targeted developer accounts. The project did not have any recent updates in 5 years until the new malicious version, that was used to download and launch a trojan. The malicious version was taken down by the Pypi team [23].
  In conclusion, our markers align, and the applicable ones are: $\boldsymbol{H_2, H_5, H_7}$

To conclude, since we could not find libraries that were experiencing hostile takeover at that moment, we decided to only test libraries from the first category (susceptible for a takeover), and *"exotel"* from the second category in the toy application: *"crcmod, sgmllib3k, oauth2client, cachy, flask-babel, toml, pytest-forked, (+ exotel)"*.

Hypothesis
Our conclusion is that libraries mentioned above are susceptible to a hostile takeover, and we were interested in whether Nexus IQ would provide any warning to answer **RQ3**. We already knew that for *"crcmod, sgmllib3k, oauth2client, cachy, flask-babel, toml"* projects there was no warning since we have already tested them. That is why the main focus was on *"pytest-forked"* and *"exotel"*.

Results
The result aligned with our initial hypothesis, Nexus did not provide any warnings for the libraries in question. We already knew that it would not provide anything for abandoned and insufficiently maintained ones, but after looking at the Nexus report, we learned that it does not warn for other libraries that are susceptible for take over (*"pytest-forked"*), or for the ones that were taken over in the past (*"exotel"*).

5.6 Vulnerable dependencies Nexus IQ Test

In this section, we will choose which vulnerable libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection

To select appropriate libraries, we searched for the ones that had a known vulnerability, i.e. an assigned CVE.

- **httplib2 == 0.17.0** - Versions prior to version 0.18.0 have two CVEs *("CVE-2021-21240, CVE-2020-11078")*, which may cause Denial of Service and allow the attacker to change request headers and body. Their scores are 7.5 and 6.8. [3]
  Therefore, all of the markers apply: $\boldsymbol{V_1, V_2, V_3}$.
- **numpy == 1.13.0** - Older versions have two CVEs *("CVE-2017-12852", CVE-2021-34141")*, with invalid input validation and incorrect copying.
  Therefore, all of the markers apply: $\boldsymbol{V_1, V_2, V_3}$.
- **pillow == 9.0.0** - Versions prior to the newest (9.3.0) have multiple CVEs *("CVE-2022-24303, CVE-2022-45198, CVE-2022-45199, CVE-2022-30595")*.
  Therefore, all of the markers apply: $\boldsymbol{V_1, V_2, V_3}$.

Hypothesis

Our conclusion is that libraries mentioned above are vulnerable, and we are interested whether Nexus IQ would detect every CVE and the correct CSS score in order to answer **RQ3**. Based on Sonatype's documentation examined in chapter 3.4.1, we hypothesized that Nexus would provide correct warnings.

Results

After successful scan of the toy application, the results aligned with our initial hypothesis. Nexus IQ found all of the vulnerabilities in question with the correct CSS score. Figure 11 below shows an example of a Nexus IQ warning.

httplib2 (py3-none-any) 0.17.0 (.whl)

THREAT / POLICY NAME	ACTION	CONSTRAINT	CONDITION
9 Security-High	Fail	CVSS >=7 and <10	Found security vulnerability CVE-2021-21240 with severity >= 7 (severity = 7.5) Found security vulnerability CVE-2021-21240 with severity < 10 (severity = 7.5) Found security vulnerability CVE-2021-21240 with status 'Open', not 'Not Applicable'
7 Security-Medium	Warn	CVSS >=4 and <7	Found security vulnerability CVE-2020-11078 with severity >= 4 (severity = 6.8) Found security vulnerability CVE-2020-11078 with severity < 7 (severity = 6.8) Found security vulnerability CVE-2020-11078 with status 'Open', not 'Not Applicable'

Figure 11: Nexus IQ "httplib2 0.17.0" report

5.7 Typosquatting Nexus IQ Test

In this section, we will choose which typosquat libraries to examine. Unfortunately, it will not be possible to use them in an application since they contain malware.

Library selection

In order to select appropriate libraries, we searched for known typosquatting examples. More

precisely, we searched for libraries that imitated ones from our subset of 1000 most popular packages. All of the selected libraries in the list below contained malware that targeted developers and their cryptocurrency, and were already downloaded a few hundred times, hours after they were published.

- **baeutifulsoup4, beautifulsup4** - Imitates a popular library *"beautifulsoup4"*, that had 43,446,219 downloads in September. Such an immense number creates a significant potential blast radius for the attacker, to take advantage of the developer typos.
  Therefore, all of the markers apply: $\boldsymbol{T_1, T_2, T_3}$.
- **crpytography, cryptograpyh** - Imitates a popular library *"cryptography"*, that had 138,618,886 downloads in September.
  Therefore, all of the markers apply: $\boldsymbol{T_1, T_2, T_3}$.
- **djangoo** - Imitates a popular library *"django"*, that had 7,711,076 downloads in September.
  Therefore, all of the markers apply: $\boldsymbol{T_1, T_2, T_3}$.
- **pillwo** - Imitates a popular library *"pillow"*, that had 42,692,436 downloads in September.
  Therefore, all of the markers apply: $\boldsymbol{T_1, T_2, T_3}$.

The examples were acquired from Phylum blog [32]. At the time, the article was only 12 days old and all of the examples were already taken down by Pypi. Nevertheless, examples like these always continue to reemerge even after being taken down.

Hypothesis
Our conclusion is that previously mentioned libraries are malicious typosquatting examples, and we are interested in whether Nexus IQ would provide any warnings to answer **RQ3**. Based on Sonatype's documentation examined in chapter 3.4, we hypothesized that Nexus IQ would not find typosquatting examples, when used without Nexus Firewall. It is powered by Nexus Intelligence and is the only Nexus tool that actually blocks malicious packages from being downloaded into the organisations repository. Therefore, by using only Lifecycle and Repository (Rabobank's Nexus IQ), we hypothesized that a malicious component could be downloaded.

Results
Unfortunately, we were not able to test this attack because, as previously mentioned, typosquatting examples contain malware. Certain malicious libraries have an install script which executes malicious code even just by installing the package. To test this safely, we would have to set up a malware analysis lab and probably completely reset the machine, which is out of scope of this paper.

5.8 Dependency confusion Nexus IQ Test

In this section, we will simulate a dependency confusion attack. We will firstly investigate if it was successful, and if so, whether Nexus IQ would provide any warnings.

Library selection
The library selection for this attack was quite different from the other attack categories, since we had to search for an internal Rabobank library instead of a public one. Firstly, we checked the internal Rabobank Nexus Repository to find an existing python library; the chosen library was **"category-new"**, with the highest version of *"5.0.0.20220610.11"*. Then, we created a project with the same name and a higher version, and published it on Pypi; **"category-new == 7.2.3"**. Figure 12 shows how the project looks like on Pypi.

Figure 12: Category-new (Pypi)

The contents of the code were not important (a simple hello world function), since we were only interested which version is going to be installed in order to check if the confusion attack worked.

Hypothesis
Our conclusion is that **"category-new == 5.0.0.20220610.11"** is now susceptible to dependency confusion attack, and we are interested whether it would succeed and if Nexus IQ would provide any warnings in order to answer **RQ3**. At this point, we were unable to provide an initial hypothesis, since the success of the attack heavily depends on the way a certain package manager is configured.

Results
The results were quite surprising; the dependency confusion attack was successful, and Nexus did not provide any warnings. To be more specific, the attack depended on the way that *"pip.conf"* file was configured. The file uses two different variables, *"index-url"* and *"extra-index-url"*. We configured the file in a way that the first index *(index-url)* checks the internal repository, and the second index *(extra-index-url)* checks the Pypi repository through the Nexus proxy. The following indexes illustrate the configuration, however, they are intentionally inaccurate because of privacy and security concernes, the purpose is only to aid in understanding the general idea.

- index-url = https://nexus/..../internal-rabobank-repository/ *(first one)*
- extra-index-url = https://nexus/.../pypi-proxy/ *(second one)*

With this configuration, the course of events should have been that the build agent **(?)** firstly checks the internal repository, and only if the package is not found, checks the public Pypi repository through the proxy. However, this course of event did not happen and instead, the public *"category-new"* package always got installed first, since it had a higher version. Therefore, the dependency confusion attack worked. Furthermore, we tested the attack for different combinations of indexes and it worked for all cases except one. The case where the attack did not work was where both of the indexes pointed only to the internal repository. However, in that case it is not possible for the developer to fetch any public packages from Pypi, which makes it extremely unpractical and unlikely to be used.

Moreover, we found possible countermeasures, that are available when using Nexus products. First one is Nexus Firewall, and it provides an option to defend against dependency confusion attacks by simply checking *"Components in this repository count as proprietary for dependency confusion attacks"* option. However, Rabobank does not use the Firewall tool, which is perhaps something to consider in the future. The second countermeasure is possible with Nexus Repository (which Rabobank does use) through the manipulation of certain routing rules [36]. However, we did not get a clear answer if this is used, or if they were even aware of it.

5.9 Transitive dependencies Nexus IQ Test

In this section, we will choose which transitive libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection

Based on previous results, we decided to only choose vulnerable libraries as transitive ones, since we saw that Nexus did not provide any warnings for the other categories. We chose **"iospy-tools"** library because it had 19 non-upgraded transitive dependencies and certain ones contained vulnerabilities. The outdated dependency list was: *" autopep8==1.5.3, bsdiff4==1.1.9, certifi==2020.6.20, chardet==3.0.4, flake8==3.8.3, humanize==2.5.0, idna==2.10, importlib-metadata==1.7.0, mccabe==0.6.1, progressbar==2.5, pycodestyle==2.6.0, pyflakes==2.2.0, pyusb==1.0.2, remotezip==0.9.2, requests==2.24.0, tabulate==0.8.7, toml==0.10.1, urllib3==1.25.9, zipp==1.2.0"*, where all of them had newer versions.

Hypothesis
Our conclusion is that previously mentioned library has vulnerable transitive dependencies, and we are interested whether Nexus IQ would provide any warnings in order to answer **RQ3**. Based on previous results, we hypothesized that Nexus will provide warnings for vulnerable transitive dependencies.

Results
After successful scan of the toy application, the results aligned with our initial hypothesis. Nexus IQ found all of the vulnerable transitive dependencies. Furthermore, it gave a warning that new versions are available **???**. Figure 13 shows how the summary of the build report looks like.

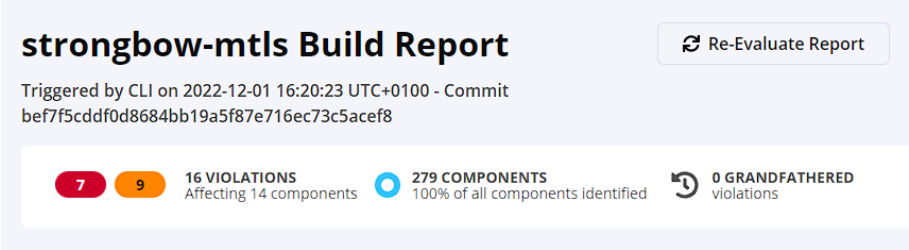

Figure 13: Nexus IQ summary report

Furthermore, Nexus also creates an SBOM with all the direct and transitive dependencies, figure 14 shows an example of it. It is important to highlight that the list is actually quite big, so the figure shows only the first few dependencies.

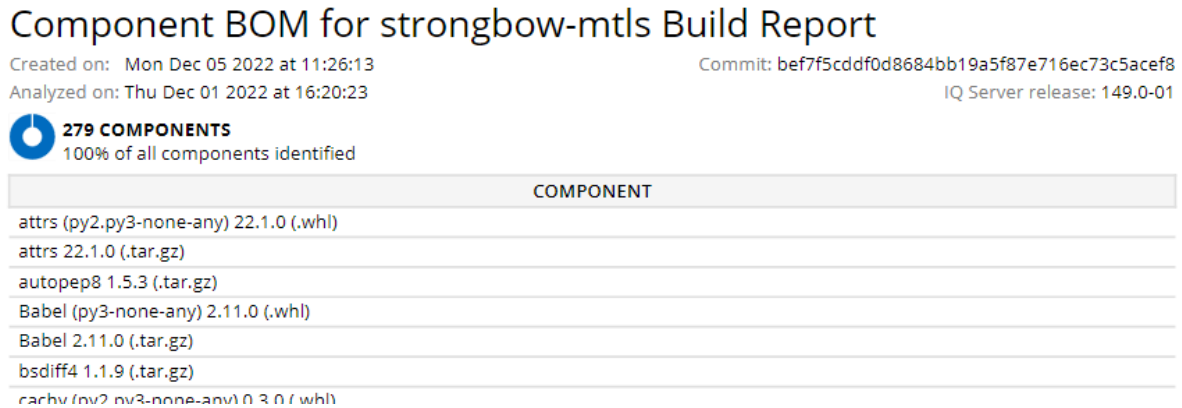

Figure 14: Nexus IQ SBOM report

6 Java Application Test Using Nexus IQ and Maven

This chapter describes a similar investigation as chapter 5. The difference is that the investigation is based on Maven packages, which are used in Java language, instead of Pypi packages used for Python. We hypothesized that the results would align with the results of PyPi analysis in chapter 5, which showed that Nexus IQ essentially alerts only on vulnerable libraries. Nevertheless, we decided to inspect it in detail, since Sonatype maintains Maven repository and thus, Nexus could contain more information for Maven packages, which could result in a different outcome.

Similarly to Chapter 5, we will create seven toy applications that use potentially dangerous libraries from each of the seven attack-vector categories, and test them with Nexus IQ in order to investigate whether it would find and report any issues to answer **RQ3** (*"For which attack vectors does Nexus IQ provide an alert?"*). The reasoning behind why and how certain libraries were chosen is explained in sections 6.1 and 6.2, and the paragraph "Library selection", which can be found at the beginning of each attack-vector analysis sub-chapter. The results of the Nexus IQ test can be found in the "Results" paragraph which is the final paragraph of each attack-vector analysis sub-chapter. Also, it is important to mention that every time we refer to Nexus IQ, we mean Nexus Lifecycle and Nexus Repository since those are the solutions that Rabobank uses.

6.1 Maven Data Collection

The process of data collection was much more challenging than for the PyPi packages. Maven repository does not provide a lot of public information about the package, such as download count, name of the author, maintainers, etc., which is why we had difficulties in collecting top 1000 Maven packages. That is why we used an already finished project, that collects top 1000 packages, which are sorted by download popularity as of September 2022 [51]. However, it was unclear how the author obtained the data and we could not confirm whether it was the correct "ranking". Nevertheless, we decided to use the mentioned list since, for the purpose of our investigation, it is not crucial that the packages are explicitly in the top 1000. It is enough that we have a data sample to choose the examples from, and that the packages are relatively popular.

Similarly as in chapter 5.1, we wanted to scrape relevant information about packages from the "top" 1000 list. However, as already mentioned, it was difficult to obtain almost any information about a package from Maven repository, which is why we decided to collect the data from *Libraries.io* website. It contains a lot of useful information about every public package, gathered from various public repositories. In addition, it provides an API [8], which made the collection much easier. An example of collected data for one package can be seen in Figure 15.

```
dependent_repos_count: 4037
dependents_count: 1968
deprecation_reason: null
forks: 35
homepage: "https://codehaus-plexus.github.io/"
latest_release_number: "3.5.0"
latest_release_published_at: "2022-10-23T08:42:04.000Z"
latest_stable_release_number: "3.5.0"
latest_stable_release_published_at: "2022-10-23T08:42:04.000Z"
name: "org.codehaus.plexus:plexus-utils"
rank: 21
stars: 23
status: null
num_of_versions: 76
num_of_contributors: 30
any_outdated_dependencies: 0
is_deprecated: 0
is_unmaintained: 0
```


Figure 15: Collected data of "plexus-utils" project

Most of the data is quite self-explanatory, so we will explain only the vague parts in more detail.

- **"forks"** and **"stars"** - collected from GitHub
- **"rank"** - does not refer to Maven rank, instead, it is an *Libraries.io* algorithm to index search results, where the maximum score is 30 points. The code is added or subtracted based on five categories; Code, Community, Distribution, Documentation and Usage. A higher score should implicate a "healthier" package. [12]
- **"num_of_contributors"** - the maximum number of contributors that *Libraries.io* collects is 30. This means that the actual number could be higher.
- **"status"** - can be "active", "deprecated", "removed", or "null" (not defined)

Similarly to chapter 5.1, we decided to further analyse this data in order to choose appropriate libraries for the toy application.

6.2 Maven Data Analysis

This chapter analyzes the collected data from 6.1 in order to guide our *"Library selection"* in the following chapters. It groups the data on certain interesting features, which are inspired by the markers from chapter 4.

Figure 16 shows how many packages (out of the 1000) did not have a new release since N years ago.

Latest release older than:	*One year ago (2021)*	*Two years ago (2020)*	*Three years ago (2019)*	*Four years ago (2018)*	*Five years ago (2017)*	*>six years ago (2016-2005)*
Number of packages:	*83*	*35*	*31*	*22*	*20*	*114*

Figure 16: Number of packages in respect to their latest release date

It is interesting to see that amongst these 1000 projects, many of them do not have an up-to-date latest release (according to previously mentioned markers). To be more specific, 222 projects did not have an updated version for more than two years. However, 83 projects had the last updated version one year ago, and 695 had the last one less than a year ago.

Figure 17 shows five of the oldest identified projects (no new release in "N" years) in descending order, together with the number of projects that currently depend on them.

Package name:	*"xmlenc:xmlenc"*	*"stax:stax-api"*	*"classworlds:classworlds"*	*"sslext:sslext"*	*"oro:oro"*
Latest release date:	*24.11.2006.*	*31.08.2006.*	*12.01.2006.*	*08.11.2005.*	*08.11.2005.*
Dependant repositories:	*411*	*944*	*129*	*29*	*2249*

Figure 17: The oldest identified projects

It is quite alarming to see that these projects did not have a new version in over 16 years, especially since a few thousand repositories depend on them. Furthermore, it is interesting to mention that none of them were marked as abandoned.

Figure 18 shows how many projects have N number of contributors. It is important to highlight that the maximum number of contributors collected through *Libraries.io* API is 30, so certain projects might have more.

Number of contributors:	*N >= 30*	*30> N >=20*	*20> N >=10*	*10> N >=5*	*5> N >0*	*N = 0*
Number of packages:	*596*	*63*	*67*	*42*	*21*	*211*

Figure 18: Number of packages in respect to the number of contributors

It is quite positive to see that 59.6% of packages have 30 or more contributors, but quite concerning that 21.1% have zero. However, the 21.1% might not be completely accurate, since it is possible that *Libraries.io* could not collect the number of contributors (e.g. no homepage, no associated GitHUb, project deleted, etc.).

Figure 19 shows how many packages have a certain development status.

Development status:	*"Active"*	*"Removed"*	*"Deprecated"*	*"None"*	*" "*
Number of packages:	*5*	*2*	*7*	*958*	*28*

Figure 19: Number of packages in respect to development status

It is not possible to draw any conclusion from figure 19, since the majority of the packages do not have a defined development status.

Figure 20 shows five most popular packages (out of the 1000) based on the number of dependant repositories, together with their latest release date.

Package name:	*"junit:junit"*	*"mysql:mysql-connector-java"*	*"'org.springframework.boot:spring-boot-starter-web"*	*"org.springframework.boot:spring-boot-starter-test"*	*"org.slf4j:slf4j-api "*
Dependant repositories:	*408,330*	*152,731*	*142,121*	*136,652*	128,362
Latest release date:	*13.02.2021.*	*08.03.2022.*	*24.11.2022.*	24.11.2022.	*25.11.2022.*

Figure 20: Most popular packages based on the number of dependant repositories, with their latest release date

The most popular package found (*"junit:junit"*) has almost half a million dependant repositories, but did not have a new release in more than one year. Based on that marker, we could argue that it is not sufficiently maintained, so we will examine it more closely in the following chapter (6.5). However, the other 4 packages have been recently updated, which could indicate that they are properly maintained.
Additional interesting information that can aid in the library selection sections in the following chapters:

- Number of packages with **outdated dependencies**: 63 - this suggests that 63 packages might belong to vulnerable, insufficiently maintained or even abandoned category
- Number of packages with only **one version**: 13 - this suggests that 13 packages might be abandoned, unless they are completely new
- Number of packages that have **more than 1000 stars**, with the latest release **older than 2 years**: 33 - this suggests that 33 packages might belong to insufficiently maintained, abandoned, or even susceptible to hostile takeover category

6.3 Abandonware Nexus IQ Test

In this section, we will choose which abandoned libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
We decided to firstly investigate packages found in *Figure* 17, since they are an extreme case of the latest release date and, therefore, a good indicator that the packages are indeed abandoned.

Secondly, we looked into the development status assigned to each of those five packages. Interestingly, none of them where marked as deprecated, even though they did not have a new release in over 16 years. That is why, on top of the two oldest projects from *Figure* 17, we decided to also include two additional projects with the development status marked as deprecated; *"org.sonatype.aether:aether-util"* with 294 dependant repositores and *"asm:asm"* with 4585 dependant repositories. The list below shows chosen projects:

- **oro:oro** - There was no new release in 17 years, no homepage found, therefore, also no issues can be posted or closed. It does not have any dependencies, so it did not depend on non-upgraded ones.
  Therefore, the applicable markers are: $\boldsymbol{A_1, A_3, A_4}$.
- **sslext:sslext** - There was no new release in 17 years, no homepage found, therefore, also no issues can be posted or closed. It depends on a non-upgraded library, which also does

not have a licence ("struts:struts == 1.2.7").
Therefore, the applicable markers are: $\boldsymbol{A_1, A_3, A_4, A_6}$.

- **asm:asm** - No new release in 9 years, homepage is not reachable. It is annotated as deprecated at *Libraries.io* and provides a redirect to the *"org.ow2.asm:asm"* package. Moreover, it does not have any dependencies.
  Therefore, the applicable markers are: $\boldsymbol{A_1, A_3, A_4, A_5 A_7}$.

- **org.sonatype.aether:aether-util** - No new release in 11 years, homepage could not be reached. GitHub homepage was found, but it is archived, annotated as deprecated both at GitHUb and *Libraries.io*. Moreover, it provides a redirect to the *"org.eclipse.aether:aether-util"* package and it does not have any dependencies.
  Therefore, the applicable markers are: $\boldsymbol{A_1, A_2, A_3, A_4, A_5 A_7}$.

The last two packages were already confirmed to be abandoned (deprecated development status), so the fact that a lot of our markers apply further proves that the choice was good. It is also important to highlight that we did not check if the owner is reachable (A_9).

Hypothesis
Our conclusion is that the above libraries are abandoned and we are interested in whether Nexus IQ would detect it in order to answer RQ3 (*"For which attack vectors does Nexus IQ provide an alert?"*). Based on the previous conclusions from 5.3, we hypothesized that Nexus would detect certain vulnerabilities, since the packages are not updated for a long time and could depend on non-upgraded dependency versions, but that it would not warn us that the packages were indeed abandoned.

Results
The results aligned with our hypothesis. Similarly to abandonware analysis for PyPi packages in section 5.3, Nexus IQ did not provide any warnings about the abandoned projects. It only provided warnings regarding vulnerabilities within those packages, or in their outdated transitive dependencies. Therefore, Nexus IQ is not capable of detecting abandoned libraries.

6.4 Insufficiently maintained libraries Nexus IQ Test

In this section, we will choose which insufficiently maintained libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
To identify insufficiently maintained libraries, we used an intersection between collected data from *Figure 16* and *Figure 18*. That means we only searched for libraries with the latest release date older than a year, and a low number of contributors. Moreover, we decided to further concentrate on projects that belong to the "additional interesting information" list from chapter 6.2. This means that we also searched for popular projects (more than 1000 stars) and projects with outdated dependencies. This approach aligns with the previously stated markers (4.5) and gives a good indication that the libraries are indeed insufficiently maintained. The list below shows chosen projects:

- **io.jsonwebtoken:jjwt** - No new release in 4 years, some new commits and closed issues in the last year on GitHub, 30 or more contributors, depends on an outdated dependency version ("com.fasterxml.jackson.core:jackson-databind == 2.9.6").
  Therefore, the applicable markers are: $\boldsymbol{I_1, I_3, I_4, I_9, I_{11}}$.

- **org.dom4j:dom4j** - No new release in 2 years, no new commits in 2 years, less than half of the issues closed in the past year, 5 contributors, depends on two outdated dependency

versions.
Therefore, the applicable markers are: $\boldsymbol{I_1, I_2, I_3, I_4, I_9, I_{10} I_{11}}$.

It is important to higlight that we did not check the last two markers (I_{12} and I_{13}), since we could not get information on the author from *Libraries.io*.

Hypothesis
Our conclusion is that the libraries mentioned above are insufficiently maintained and we are interested whether Nexus IQ would detect it in order to answer RQ3. Based on the results from chapter 5, we hypothesized that Nexus would not warn us that the libraries are not sufficiently maintained.

Results
Our hypothesis aligned with the result, since there were no warnings that the libraries above were insufficiently maintained. Therefore, Nexus IQ is not capable of detecting insufficiently maintained libraries.

6.5 Hostile takeover Nexus IQ Test

In this section, we will choose which libraries to use in our toy application. The selection will differ from the selection for PyPi hostile packages 5.5, since Maven is a more secure repository and it was not possible to find known examples of hostile takeover (which does not mean it does not happen). That is why the library selection will focus only on the libraries that are susceptible to hostile takeover. Then, we will scan the application with Nexus IQ and examine the results.

Library selection
According to the markers H_1 and H_5, libraries that have a higher chance of being taken over are very popular ones since it creates a bigger attack surface, and abandoned or insufficiently maintained ones since the takeover remains unnoticed for a longer time period. We firstly decided to look at *"junit:Junit"* package since it is the most popular one from our list, based on the number of dependant repositories, and it did not have a new release in over a year ago. This could indicate that it is not sufficiently maintained and is susceptible for a hostile takeover, especially since it is very popular (408,330 dependant repositories). Then, we selected *"org.pegdown:pegdown"* since it seems abandoned (last release is in 2015, contains outdated dependencies), and it is quite popular (1263 stars on GitHub, and 858 dependants). Similarly, we chose *"com.googlecode.json-simple:json-simple"* (latest release date in 2012, contains outdated dependencies and has 15,242 dependant repositories), and *"commons-fileupload:commons-fileupload"* (latest release date in 2018, contains non updated dependencies and has 43800 dependant repositories).

According to the marker H_4, if the project's domain is expired and the attacker acquires it, he could take over the project. However, we concluded that this is extremely difficult, if not, even impossible for Maven repository, since it enforces a stronger validation process and obtaining an email/password combination should not be enough. To be specific, when deploying the project (or a new version), the author has to include the *settings.xml* file which has to be signed with the author's private key, contain his username and a passphrase [13]. Furthermore, the attack gets even harder since Sonatype states that they analyze which domains are abandoned, and then, they claim them [14]. Therefore, it is not possible to take over Maven projects by a "simple" domain hijack, only through a more sophisticated attack or a legitimate ownership transfer to a malicious user.
Hence, the selected libraries are: **junit:Junit, org.pegdown:pegdown, com.googlecode.json-simple:json-simple, commons-fileupload:commons-fileupload**.

Hypothesis

Our conclusion is that libraries mentioned above are susceptible to a hostile takeover, and we were interested in whether Nexus IQ would provide any warnings, to answer **RQ3**. Based on the results of PyPi analysis 5.5, we hypothesized that it would not provide any warnings.

Results
The result aligned with our initial hypothesis, since there were no warnings that the libraries above are susceptible to a hostile takeover. Therefore, Nexus IQ is not capable of detecting hostile takeover susceptible libraries.

6.6 Vulnerable dependencies Nexus IQ Test

In this section, we will choose which vulnerable libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
To select appropriate libraries, we searched for the ones that had a known vulnerability, i.e. an assigned CVE.

- **struts:struts==1.2.7** - Versions prior to 1.2.9 allow remote attackers to bypass validation via a request with a "org.apache.struts.taglib.html" (CVE-2006-1546). Addditionally, we found six more CVE's associated to this version [10].
  Therefore, all of the markers apply: $\boldsymbol{V_1, V_2, V_3}$.
- **com.fasterxml.jackson.core:jackson-databind==2.13.2.1** - In versions prior to 2.14.0-rc1, resource exhaustion can occur because of a lack of a check in primitive value deserializers (CVE-2022-42003). Additionally, in versions prior to 2.13.4, resource exhaustion can occur because of a lack of a check in "BeanDeserializer.deserializeFromArray" (CVE-2022-42004) [11].
  Therefore, all of the markers apply: $\boldsymbol{V_1, V_2, V_3}$.

Hypothesis
Our conclusion is that libraries mentioned above are vulnerable, and we are interested in whether Nexus IQ would detect every CVE and the correct CSS score in order to answer **RQ3**. Based on the PyPi analysis from section 5.6, we hypothesized that Nexus would provide correct warnings.

Results
After the successful scan of the toy application, the results aligned with our initial hypothesis. Nexus IQ found all of the vulnerabilities in question with the correct CSS score. Figure 21 shows an example of a Nexus IQ warning.

com.fasterxml.jackson.core : jackson-databind : 2.13.2.1

THREAT / POLICY NAME	ACTION	CONSTRAINT	CONDITION
9 Security-High	Fail	CVSS >=7 and <10	Found security vulnerability CVE-2022-42003 with severity >= 7 (severity = 7.5) Found security vulnerability CVE-2022-42003 with severity < 10 (severity = 7.5) Found security vulnerability CVE-2022-42003 with status 'Open', not 'Not Applicable'
9 Security-High	Fail	CVSS >=7 and <10	Found security vulnerability CVE-2022-42004 with severity >= 7 (severity = 7.5) Found security vulnerability CVE-2022-42004 with severity < 10 (severity = 7.5) Found security vulnerability CVE-2022-42004 with status 'Open', not 'Not Applicable'

Figure 21: Nexus IQ "jackson-databind 2.13.2.1" report

6.7 Typosquatting Nexus IQ Test

In this section, we will choose which typosquatted libraries to examine. Unfortunatelly, it will not be possible to use them in an application since they contain malware.

Library selection
In order to select appropriate libraries, we searched for known typosquatting examples. However, it is important to highlight that typosquatting attacks are more difficult to be executed on Maven repository since it is built upon a strong namespacing convention. Every artifact consists of minimally three parts. First one is Group ID which is a reverse of the according domain name (e.g. "org.apache.maven"). Maven enforces the publisher to prove the ownership of either the DNS, or the repository such as GitHub. It is followed by an Artifact ID, which is the given name of the library, and a Version ID, which is the current version of the project. Group ID is the most important part, since it prevents the attacker from impersonating the package name. The attacker cannot prove the ownership of the same domain, nor he can create an extremely similar one (one to few letters different), since Maven checks for that as well. Therefore, in repositores witouth a namespace convention (such as PyPi), it is easier to trick a user into using a typosquatted packages such as "foo-bar" instead of a legit one "fooBar". An example of typosquatting attempt in Maven, such as *"com.github.codingandcoding:maven-compiler-plugin"* is clearly different from the legitimate one *"org.apache.maven.plugins:maven-compiler-plugin"* [47]. However, the typosquatting attacks still occur, and certain example libraries can be found below.

- **com.github.codingandcoding:maven-compiler-plugin:3.9.0** - Impersonates legitimate *"org.apache.maven.plugins:maven-compiler-plugin"* package, and it had only one version (3.9.0). Contains malicious code injected within *CompilerMojo* class and is capable of executing arbitrary code downloaded from the C2 server [47]. The package was removed by Sonatype.

- **com.github.codingandcoding:mail-watcher-plugin:1.16, 1.17** - Impersonates legitimate *"org.apache.maven.plugins:mail-watcher-plugin"* package, ant it had two versions. Similarly, the malicious code was found in *send()* method and contained a different hardcoded server [47]. The package was removed by Sonatype.

- **com.github.codingandcoding:servlet-api:3.2.0** - Impersonates multiple legitimate packages, that are published under identical names (but different group IDs) by trustworthy projects like Apache, Eclipse, Tomcat, etc. Again, it had only one version, malicious code, and was removed by Sonatype [47].

Hypothesis
Our conclusion is that previously mentioned libraries are malicious typosquatting examples, and we are interested in whether Nexus IQ would provide any warnings in order to answer **RQ3**. Based on Sonatype's documentation examined in chapter 3.4, we hypothesized that Nexus IQ would not find typosquatting examples, when used without Nexus Firewall. It is powered by Nexus Intelligence and is the only Nexus tool that actually blocks malicious packages from being downloaded into the organisations repository. Therefore, by using only Lifecycle and Repository (Rabobank's Nexus IQ), we hypothesized that a malicious component could be downloaded.
Results
Unfortunately, we were not able to test this attack because, as previously mentioned, typosquatting examples contain malware. Certain malicious libraries have an install script which executes malicious code even just by installing the package. To test this safely, we would have to set up a malware analysis lab and probably completely reset the machine, which is out of scope of this paper.

6.8 Dependency confusion Nexus IQ Test

In this section, we will not simulate a dependency confusion attack as we did in 5.8. The reason behind it is that it would be too challenging because of Maven's strong namespace conventions. Therefore, this chapter is present only for completness.

Library selection
As already mentioned, Maven enforces strong naming conventions which is why the simulation for dependency confusion proved to be too challenging in the scope of this research. We would have to purchase a domain with the same name as the internal Rabobank dependency (provided that it is available) and prove that we are the owner. Moreover, Sonatype demands a lot of additional requirements when publishing a package, which can be found in their documentation [44].

Hypothesis
It is hard to hypothesize whether this attack would be successful or not, since it heavily depends on the way package manager and build agent work. However, based on the PyPi dependency confusion results (5.8), we do hypothesize that if the attack is successful, Nexus IQ would not provide any warnings.

Results
Unfortunately, we were not able to test this attack because, as previously mentioned, the process of publishing an artifact on Maven with a specific name and Group ID proved to be out of scope of this research. However, we advise Rabobank to look into this in depth.

6.9 Transitive dependencies Nexus IQ Test

In this section, we will choose which transitive libraries to use in our toy application, scan the application with Nexus IQ and examine the results.

Library selection
Based on previous results, we decided to only choose vulnerable libraries as transitive ones, since we saw that Nexus did not provide any warnings for the other categories. We choose **"sslext:sslext:1.2.0"**, since it had a vulnerable transitive dependency *"struts:struts:1.2.7"*, and **"io.jsonwebtoken:jjwt-jackson:0.11.5"** since it had *"com.fasterxml.jackson.core:jackson-databind:2.13.2.1"* as a vulnerable transitive dependency.

Hypothesis
Our conclusion is that previously mentioned libraries have vulnerable transitive dependencies, and we are interested whether Nexus IQ would provide any warnings in order to answer **RQ3**. Based on the previous results from section 5.9, we hypothesized that Nexus will provide warnings for vulnerable transitive dependencies.

Results
After successful scan of the toy application, the results aligned with our initial hypothesis. Nexus IQ found all of the vulnerable transitive dependencies. Moreover, it found another vulnerability in the *"com.fasterxml.jackson.core:jackson-core:2.13.2"* package, which is a two level deep transitive dependency of *"io.jsonwebtoken:jjwt-jackson:0.11.5"* (Figure 22).

com.fasterxml.jackson.core : jackson-core : 2.13.2

THREAT / POLICY NAME	ACTION	CONSTRAINT	CONDITION
9 Security-High	Fail	CVSS >=7 and <10	Found security vulnerability sonatype-2022-6438 with severity >= 7 (severity = 7.5) Found security vulnerability sonatype-2022-6438 with severity < 10 (severity = 7.5) Found security vulnerability sonatype-2022-6438 with status 'Open', not 'Not Applicable'

Figure 22: Nexus IQ "jackson-core 2.13.2" report

The vulnerability was labeled with a Sonatype ID instead of CVE, which means that the vulnerability is still not CVE identified. As already mentioned in section 3.4.1, Nexus is powered by Nexus Intelligence which means that it ingests vulnerability information from a substantial number of sources, such as GitHub advisories, blogs, feeds, etc. [48]

7 Health Score Tool

The conclusions derived from chapters 5 and 6, indicated that Nexus IQ essentially only flags vulnerable libraries, which served as a motivation for the creation of a new tool. That is why, this chapter provides a detailed explanation of a tool we created. The tool calculates the health score of any open-source package found at PyPi, Maven, npm and Conda repositories. Furthermore, it also provides explicit warnings which can indicate abandonment, insufficient maintenance, maliciousness, or possible hostile takeover through domain hijack.

7.1 Markers

In this section, we will provide a list of all the markers that are collected in order to calculate the health score, accompanied by a detailed explanation for each of them.

The tool provides a health score for any open-source library found on Pypi, Maven, npm and Conda repositories. The score is calculated based on 15 markers, derived from our previous analysis in chapter 4. The markers are listed and explained in the list below.

- **Created since** - Projects that exist for a longer period of time can indicate higher usage, could have more known vulnerabilities spotted and resolved, and are less likely to be malicious.
- **Updated since** - Projects that have not been updated for a certain amount of time could be abandoned or insufficiently maintained. (A_1, I_1)
- **Contributor count** - Higher number of contributors indicates a higher chance that the project is receiving proper updates, support, and resources. (I_{10})
- **Organisation count** - Company accounts are likely to be more trustworthy and provide proper maintenance. (I_{13})
- **Commit frequency** - Higher number of commits indicates higher maintenance. (A_3, I_2)
- **Releases count** - Higher number of releases indicates higher maintenance, less likely to be malicious, more trusted project. (T_5)
- **Closed issues count** - Higher number of closed issues indicates higher user involvement and better maintenance. (I_9)
- **Dependant projects count** - Indicates projects popularity, should be trusted more since potential vulnerabilities can be caught by more people.
- **Dependencies updated** - Outdated transitive dependencies can indicate unmaintained packages. (A_6, I_4)
- **Libraries.io rank** - It is a score provided by Libraries.io and is based on several markers. There is a slight overlap with our markers, but it is also taking into account other ones (such as is repository present, is the license present, etc.), which is the motivation behind looking at this rank as well. The higher rank (max. 32) indicates a safer package.
- **Not deprecated** - Checks if the package is marked as deprecated by the author. (A_2)
- **Maintained** - Checks if the package is marked as unmaintained by the author. (I_6)
- **Not removed** - Checks if the package is removed from the package manager.
- **Recent release** - Checks if the package had a new version in the last six months. Projects with a recent release are healthier. (I_1)

- **Homepage accessible** - When the homepage of a project is non-accessible or does not exist, it gives a strong indication that the project is abandoned. (A_3)

"Commit frequency" and "closed issues count" markers are collected from Github's API [2]. The rest of the markers are collected from the Libraries.io API (also explained in section 6.1), which is a web service that collects information on open-source packages coming from 28 different repositories [8]. Since it is "repository agnostic", meaning that it collects the same data for different packages coming from different repositories, it allows us to draw comparisons between them.

7.2 Formula

The formula used to calculate the criticality score can be seen in Figure 23 and is based on Github's project, inspired by Robin Pike's algorithm [1].

$$H_{project} = \frac{1}{\sum_i |\alpha_i|} \sum_i \alpha_i \frac{log(1 + max(S_i, T_{1,i}))}{log(1 + max(S_i, T_{2,i}))}$$

Figure 23: Health score formula

- S_i - actual value of a certain marker
- α_i - weight of a certain marker
- T_1, T_2 - lower and upper thresholds of a certain marker.

The primary distinction between the GitHub project outlined in [1] and ours is that the former focuses on identifying important projects, i.e. it assigns them a criticality score based on their importance. Meanwhile, our objective is to generate a health score for any open-source project that can be found on platforms such as Pypi, Maven, npm, and Conda. While criticality (popularity) does play a minor role in determining the health score (affects the score positively), it is not the primary focus since unpopular projects can also gain a decent score. Furthermore, we are focusing on different markers and are collecting them mainly from a different source (Libraries.io).

The logarithm function in the fraction is used to reduce potentially large numbers, and the fraction of logarithms provides a result in the 0-1 range. The maximum function in the numerator is used if we want a marker to start from a certain threshold. For example, if we assign $T_1 = 3$ for contributors, it would mean that the minimum value in the nominator would always be 3. In that case, we are only interested if the project has more than 3 contributors, and not if it has 0, 1, 2, or 3 contributors, which we consider equivalent. We did not use this threshold T_1 for any of our markers, i.e. we assigned it the value zero (so S_i is always equal or higher), but we wanted to introduce this option if a certain user wishes to have a nominator threshold. Furthermore, the maximum function in the denominator is used if we want the marker to have a certain "optimal" value. Let us take the example of contributor marker again. If we assign $T_2 = 30$, it means that we think the optimal value of the contributor marker is 30 contributors for a good package. If the real value of the contributor marker $S_{contributor}$ is for example 5, that means the denominator would have a value of $log(1 + 30)$ since 30 is higher than 5. At the same time, the value of the nominator would be $log(1 + 5)$, since the threshold is different. Now, we can see that the fraction would actually be $log(6)/log(31)$, which means we are actually dividing the nominator with the desired value of the marker. If the actual marker value $S_{contributor}$ is

the same or higher as the threshold (e.g. $S_{contributor} = 33, T_2 = 30$, the fraction will get the maximum value of 1 ($log(1 + 33)/log(1 + 33) = 1$). Moreover, the addition *"+1"* is in both logarithms in case the value of the marker is 0, since we want to avoid $log(0)$ because it is an undefined value. An example of three markers, including their weight, threshold, description, and reasoning parameters is illustrated in Figure 24. The AI weight will be explained in the later section 7.4.1.

Parameter(Si)	w(αi)	AI w(αi)	Min (T1)	Max(T2)	Description	Reasoning
created_since	1.0	9.53	0	80	Time since the project was created (in months)	Projects that exist for a longer period of time can indicate higher usage, could have more known vulnerabilities spotted and resolved, and are less likely to be malicious.
updated_since	-1.5	7.87	0	120	Time since the project was last updated (in months)	Projects that have not been updated for a certain amount of time could be abandoned or insufficiently maintained.
contributor_count	1.5	0.0	0	30	Count of project contributors (with commits)	Higher number of contributors indicates a higher chance that the project is receiving proper updates, support, and resources.

Figure 24: Parameter data

Now, if we were to use only these three markers (for the sake of simplicity), applying the formula with marker values $S_{c.since} = 30, S_{u.since} = 10, S_{contributor} = 15$ would result in the following equation.

$$H_{project} = \frac{1*log(31)/log(81)-1.5*log(11)/log(121)+1.5*log(16)/log(31)+...}{1+|(-1.5)|+1.5+...} =$$

$$= \frac{1*0.78-1.5*0.5+1.5*0.81}{4+...} \approx [0, 1]$$

Thus, the weight assigned to each marker (e.g. 1) is multiplied by the outcome of its corresponding logarithmic fraction (e.g. 0.78), which is all then divided by the sum of the absolute values of the weights. The final result is approximately in the [0,1] interval. It is approximate because one marker weight is negative, which shifts the interval a bit to the left. This means that the score can never be 1, and theoretically can go below 0; if all of the marker values are 0 and only the negative one has a value.

7.3 Warnings

In this section, we will look at all of the warnings that the tool can provide. Specifically, what do they mean and which attack categories it can flag.

Warnings
(Very <50%) **Low score** (50%-60%), indicates the package is abandoned or unmaintained.
This package has **outdated dependencies**, it might be deprecated!
This package has **no contributors**, it might be insufficiently maintained!
This package has **no previous versions**, could be malicious!
Homepage is either not provided, does not exist, or is **not reachable**! Indicates deprecation!
This package is marked as **unmaintained**/This package is marked as **deprecated**/This package is **removed** from the package manager, indicates deprecation!
This package has **not been updated for > 2 years**, indicates deprecation!
This package has **not been updated for [10-24] months**, has <**10 commits** in the last year and <**5 closed issues** in the last 3 months, might be insufficiently maintained. (3 combinations)
This package is **only <=3 months old** , could be malicious!
Maintainer domain **expiration date** is **not specified/has expired/will expire** in <60 days. Check purchasibility, possible hostile takeover attack through domain hijack!
The maintainer **domain** has a certain **status**(RedemptionPeriod,pendingdDelete...), it **will be/is available for purchase**, possible hostile takeover attack!

The table above shows shortened versions of all possible warnings the tool can provide. Abandonment and insufficient maintenance are flagged by a low score, outdated dependencies (A_6, I_4), no contributors(I_{10}), non-existing or non-reachable homepage (A_3), no recent releases (A_1, I_1), deprecation indicators (A_2, A_5, I_6), and a small number of closed issues and commits(A_4, I_2, I_9). It is important to highlight that the tool does not have malware detection, but it can flag potentially malicious packages based on no previous versions (T_5) and very fresh package (T_5) markers. For the hostile takeover attack, we are focusing only on the H_4 marker; domain hijack through expired maintainer domain. We are collecting all maintainer and author email domains for packages found only on PyPi and npm, since Maven does not provide that information (explained in more detail in section 6.1). After collection, we exclude popular email domains like gmail, yahoo, hotmail, etc., and perform a "whois" lookup on the remaining domains. The "whois" package provides several details about a domain, including its expiration date and status, which are crucial for us to determine its availability for purchase. The tool flags maintainer domains that are expired or are going to expire soon (<60 days). Furthermore, it also checks for certain domain statuses that indicate that the domain is available for purchase or will soon be. Such statuses are *RedemptionPeriod, inactive, pendingDelete, PendingRestore, locked, expired*; further explanation can be found at ICANN website [31]. In the event that a domain is available for purchase, an attacker can obtain the maintainer's precise email address, allowing them to take over their package on platforms like PyPi, particularly if 2FA authentication is not enabled. This would enable the attacker to publish a malicious version of a previously trustworthy and legitimate package, resulting in a hostile takeover; which is why it is crucial that the tool checks for expired domains.

7.4 Usage

There are two different ways the tool can be used, as a desktop application using GUI, or as a web application using REST API. In this section, we will explain both approaches.

7.4.1 Desktop application

When the tool is ran as a desktop application, the user can first see the initial GUI window. In that window, the user has the option to select one of the four different repositories (PyPi, npm, Maven and conda), and enter the name of the desired package for which the health score is to be calculated, as seen in Figure 25.

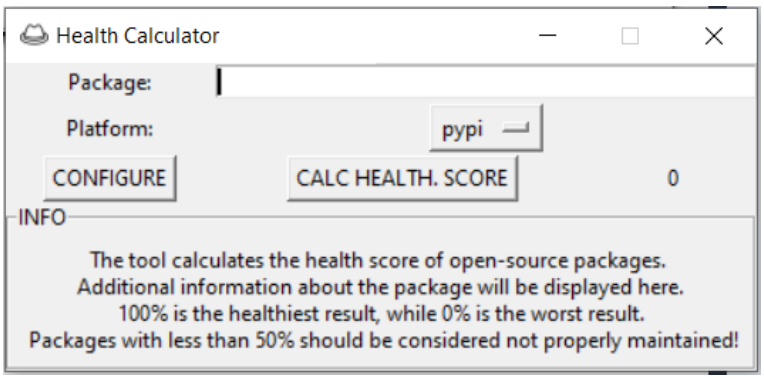

Figure 25: Initial GUI

If the provided package and repository combination does not exist, the user will get an error message that the combination does not exist. If the combination is correct, the tool will calculate and output the health score and warning messages if there are any. Therefore, the initial GUI window is used for calculating the score and for providing warning messages. The examples can be seen in Figure 26 and Figure 27.

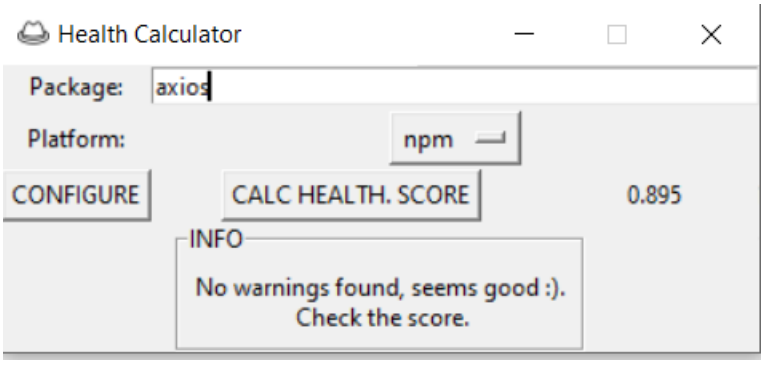

Figure 26: Health score of axios package

Figure 26 is an example of a "good" package. The health score s quite high, there are no warnings, so we can draw the conclusion that this package is safe to use, i.e. it is properly maintained.

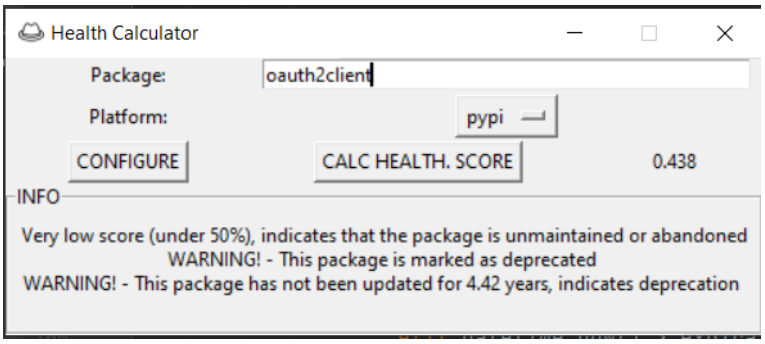

Figure 27: Health score of oauth2client package

Figure 27 is an example of an "unsafe" package, i.e. an abandoned package. The health score is lower than 50% which is a threshold for flagging insufficiently maintained or abandoned packages. Furthermore, the tool shows two additional warnings regarding deprecation which confirm that the package is indeed deprecated.

The user also has the option to click the "configure" button, which will open a new GUI window. This interface provides the formula with its explanation, together with all 15 markers, their parameters and explanations. However, the main goal of this interface is to enable the user to change certain parameters, since weights and thresholds heavily influence the result. Initially, we established default weights and thresholds that were deemed optimal after undergoing numerous tests and adjustments. Nevertheless, we aimed to grant users the liberty to modify them, as certain users may consider specific markers, such as the "updated since" marker, to be especially significant, and hence would prefer to assign a higher weight or threshold to that particular marker. Therefore, the user is free to modify any parameter, and is only required to click the save button for the adjustments to be retained. The new parameters are stored in a CSV file, which ensures that they remain preserved for the next usage, even if the user exits the program. Additionally, there is input validation implemented to prevent errors. For instance, if the user inputs letters instead of numbers, an error message will be displayed, and the changes will not be saved. An example of the "configure" interface can be seen in Figure 28.

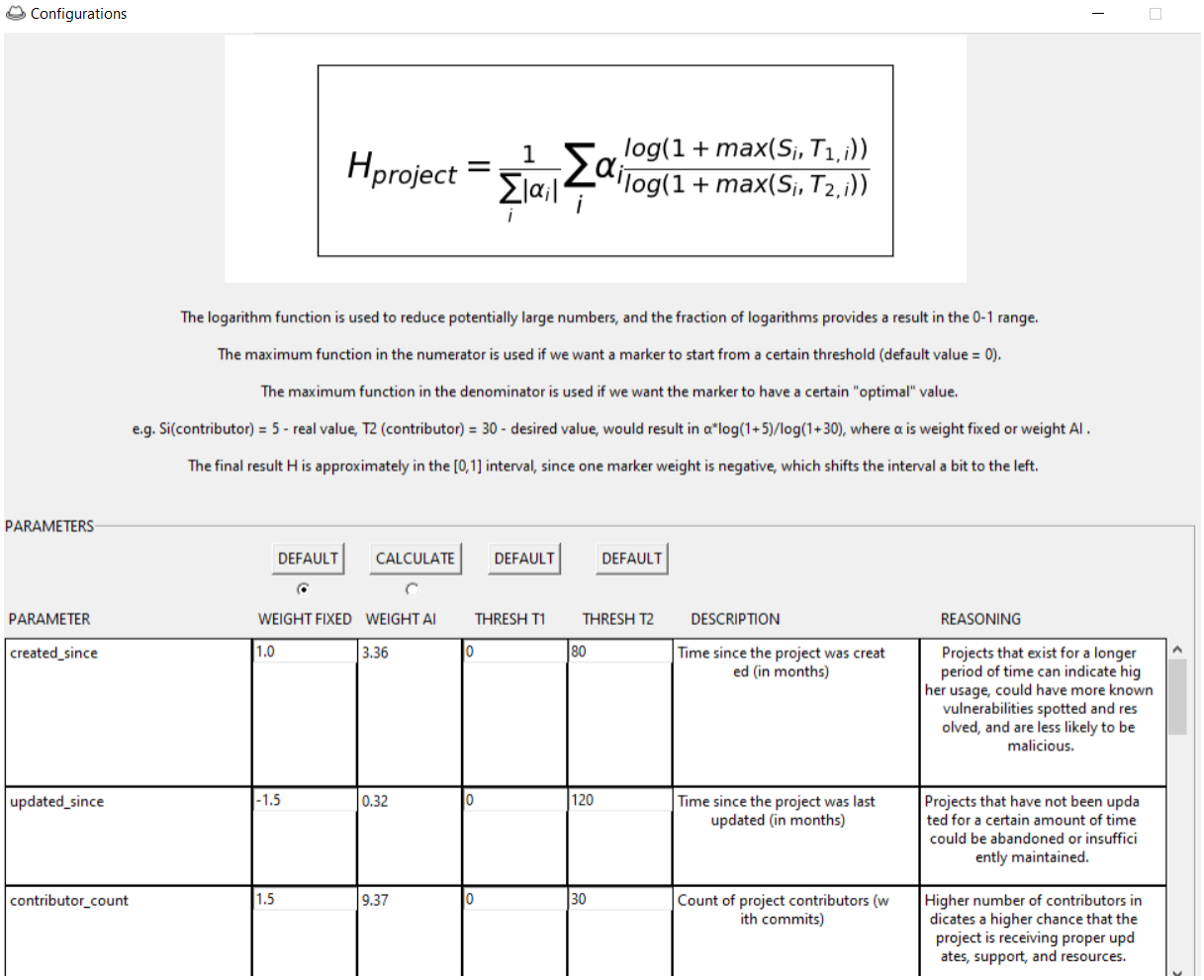

Figure 28: The "Configure" interface

From the Figure 28, we can see three "default" buttons. They enable the user to reset each parameter to its default value at any moment, in case any unwanted modifications were made. Furthermore, there is also a "calculate" button, which is used to calculate AI weights, and an according radio button "weight AI", which when selected, uses those AI weights in the calculation of the health score. The motivation behind using an AI model was to avoid subjective biases that would influence weights and thus, the health score. That is why the notion was to implement an AI model that could generate objective and optimal weights derived from a designated dataset. However, we underestimated the difficulty in constructing a sizable and high-quality dataset, which explains why the AI alternative is not functioning adequately. Every time the AI weights are computed, they are entirely different, so we can conclude that the AI weights are not functioning. Nevertheless, we left the option available, in case someone with more knowledge of AI decides to improve the dataset.

7.4.2 Server-side web application

If a user wishes to use the tool as a web application through REST application programmable interface (API), he can choose from two different endpoints.

- **GET http://localhost:5000/score/{platform}/{package}**

This endpoint returns the score of a given platform package combination. Additionally, it returns the value of each marker and accompanying warnings. Figure 29 shows an example of an endpoint for calculating the health score of "pix-diff" package, and Figure 30 shows the result.

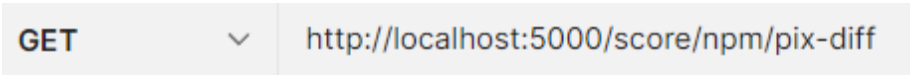

Figure 29: GET request example


```
{
  "marker_values": {
      "all_dependencies_updated": 0,
      "closed_issues_count": 0,
      "commit_frequency": 0,
      "contributor_count": 10,
      "created_since": 98,
      "dependant_projects_count": 4,
      "homepage_accessible": 0,
      "lib_rank": 14,
      "maintained": 1,
      "not_deprecated": 1,
      "not_removed": 1,
      "org_count": 2,
      "recent_release": 0,
      "releases_count": 22,
      "updated_since": 68
  },
  "score": 0.335,
  "text": "Very low score (under 50%), indicates that the package is unmaintained or abandoned\nWARNING! - This package has outdated
      dependencies, it might be deprecated \nWARNING! - Homepage is either not provided, does not exist, or is not reachable!
      Indicates deprecation! \nWARNING! - This package has not been updated for 5.67 years, indicates deprecation \n"
}
```


Figure 30: GET response example

Furthermore, there are additional **query parameters** that the user can include:

- **ai=true**
  - The fixed weights are used as a default. If the user specifies AI as true, then AI weights will be used while calculating the score. However, as previously mentioned, AI weights are not adequate.

- **w_marker = value, t1_marker = value, t2_marker**
  - e.g. w_created_since = 10, t1_contributor_count = 50, t2_org_count = 2
  These query parameters enable the user to change the value of any desired weight or threshold. However, unlike the desktop application, it does not retain changes, requiring the user to include them with each request.

The second and final endpoint is used to get information on markers, weights and thresholds, similarly to the "configure" interface of the desktop application. The response can be seen in Figure 31.

- **GET http://localhost:5000/info**


```
"all_dependencies_updated": {
    "ai_weights": 5.71,
    "description": "Checks if all transitive dependencies are up to date (0 or 1 value)",
    "reasons": "Outdated transitive dependencies can indicate unmaintained packages",
    "t1_default": 0.0,
    "t2_default": 1.0,
    "w_default": 2.0
},
```


Figure 31: GET response example

Figure 31 shows only a part of the response due to spatial limitations, i.e. it shows information only on one marker whereas normally it shows information on all 15 markers. The main goal of this endpoint is to provide the user with information on all parameters, so that the user can decide if he wishes to change certain ones in the query parameters of score endpoint.

7.5 Results

After various rounds of tests and adjustments, we successfully established appropriate default weights and thresholds for the tool. Then, we put those parameters to the test by evaluating the tool's performance on approximately 200 packages found on either Pypi, npm, Maven or conda. The outcomes of these evaluations were highly satisfactory, as the tool provided "appropriate" scores and warnings.

We began the testing phase on, to us, known "good" and "bad" packages from the Maven (6.2) and PyPi analysis (5.2) to see if the tool will provide expected results. We were pleased to discover that the tool provided high scores (over 80%) for packages already known to be "good," such as numpy, tensorflow, axios, and org.apache.maven:maven-settings. On the other hand, the tool issued low scores (mostly below 50%) and at least one warning for packages that were already known to be "bad" (again, from PyPi and Maven analyses). Based on these positive results, we concluded that the parameters were indeed appropriate.

Then, we decided to broaden the testing surface and searched for popular and widely used packages. The tool provided positively surprising results, since for certain popular and widely used packages it found warnings and provided a lower score than expected. We manually inspected markers' values of those packages, and concluded that the result was indeed correct. For example, package "express" found on npm is a very popular package with more than 250 contributors, 85,000 dependant projects, and many closed issues and commits. We expected that the score will be high and that there will be no warnings. However, the tool found that the package had outdated dependencies and gave it a score of 75%. Therefore, the tool is able to flag insufficient maintenance and provide a lower score even for very popular packages.

To conclude, based on these tests and results, we claim that the health score tool is able to flag abandoned and insufficiently maintained packages through its score and warnings. Additionally, the tool is also able to flag hostile takeover through domain hijack as well as malicious packages such as typosquatting and dependency confusion, when they have only one version or when it is a very fresh package.

7.6 Limitations

In this section, we will look into several limitations that the tool encounters, and possible workarounds.

API rate limits

As already mentioned, we are collecting 13 markers from Libraries.io API, which has a rate limit of 60 requests per minute. Since we only make three API calls per package, this limit is entirely adequate for utilizing the desktop application. However, if the tool would be used as a web application through API this limit would be too small. However, Libraries.io offers periodic releases of the full data set, and it is possible to ask for a higher rate limit, which would be a feasible workaround.

Moreover, we are collecting two of the remaining markers from GitHub API, which has a very small rate limit, of 6o requests per hour. Because of this limitation, we are only making two GitHub API calls per package, which is not enough. To be more specific, with one API call it is possible to collect maximum hundred commits from one branch within the last year. That is why, the tool searches for a main/master branch and counts the commits. However, optimally, we want to collect commits from all of the branches, which would result in many more calls (the

number of branches = number of API calls). Similarly, with the second API call it is possible to collect a maximum of 100 closed issues per package. Furthermore, even while using the tool only as a desktop application with two API calls per package, it is very easy to exceed this rate limit. However, there is a possible workaround, GitHub offers enterprise accounts which have rate limit of around 15,000 API calls per hour. The enterprise account would solve the issue of having more API calls per package, and the issue of exceeding the limit while using the tool as a desktop or web application for multiple packages.

Dependant on Libraries.io data

Another limitation is the heavy dependence on the correctness of Libraries.io data. Based on our observations, Libraries.io generally gathers accurate information regarding packages. However, we did come across a few cases where the data was incorrect, such as when the number of contributors was inaccurate or when the repository URL was empty, but the package had a GitHub repository. Moreover, if a project has repositories other than GitHub (such as gitbox), Libraries.io might choose one of those repositories instead of GitHub for the repository URL. In either of these cases, where the URL is wrongly empty or a non-GitHub repository is used, the markers for commits and closed issues will automatically be set to zero. This would result in an inaccurate health score since the tool would not even check GitHub if it believed the according repository did not exist (no URL provided).

However, Libraries.io has an open-source code base, which makes it possible to collect the data ourselves. We could inspect the code, and make certain adjustments if necessary to be sure that we are collecting correct data.

Biased weights and thresholds

We have already explained the issue of biased weights and thresholds in depth in section 7.4.1. In summary, they heavily influence the outcome of the health score result. We tried to avoid this through numerous tests and adjustments to get "appropriate" parameters. Nevertheless, this is still heavily biased since we adjusted the parameters in line with what we expected the health score result to be. Our default parameters that carry the highest weight, i.e. most importance are *updated since, dependant projects, any outdated dependencies and libraries.io rank*. However, for a different user some other markers might carry higher importance. That is why we already implemented a workaround; each user can change any parameter.

Still, those are not universal and optimal weights, which we tried to avoid by implementing an AI model that would generate those weights based on a large dataset. However, as previously explained (section 7.4.1), our attempts failed because of a lack of knowledge of AI and of creating a high-quality dataset. That is why we listed it as a limitation and hope that someone with more knowledge will continue to explore this option.

8 Future Work

In this chapter, we will discuss potential areas for improvement and future research. Specifically, it would be worthwhile to further test Nexus IQ's effectiveness against packages that contain malware, identify and address any remaining areas for improvement in the tool, and emphasize the need for repository maintainers to prioritize security measures.

Malware test with Nexus IQ
As already mentioned in Chapters 5 and 6, we were not able to test all of the attack categories with Nexus IQ. Specifically, we did not test typosquatting and hostile takeover, since those packages usually contain malware. In the scope of this research, it was not feasible to create a malware analysis lab, or our own benevolent malware package and simulate typosquatting or hostile takeover. However, from the Nexus documentation we concluded that Nexus Lifecycle would not be able to detect malware and thus, these attack categories. On the other hand, the documentation strongly claimed that Nexus Firewall (not currently used by Rabobank) has malware detection in place and is supposed to block malicious packages. An interesting avenue for future research would be to investigate the efficacy of Firewall in protecting against typosquatted and hijacked packages. It would be valuable to invite others to perform their own tests to validate the claim, that Firewall is indeed effective in this regard.

Health score tool improvements
As already mentioned in section 7.6, the tool currently has certain limitations that could be addressed to enhance its capabilities.

- **API** - Markers are collected from GitHub and Libraries.io API, which currently have a low rate limit. The tool's performance would significantly increase if the API rate limit would increase; by for example purchasing an enterprise account or asking for a higher limit. This would enable future integration and automatization of the tool within Rabobank's development pipelines.

- **Data collection** - The tool's accuracy is currently heavily dependent on the accuracy of Libraries.io data. In corporate settings like Rabobank, where the tool should be extremely precise and accurate, we recommend collecting metadata yourself and creating a dedicated database with 100% accuracy, that mirrors the information available on Libraries.io. This would enable the tool to operate with maximum efficacy and reliability.

- **AI** - Currently, the tool has an AI functionality that is inaccurate because of a small dedicated dataset. Future efforts should focus on creating a larger, high-quality dataset and thorough testing of the AI functionality. Such improvements could significantly enhance the tool's ability to accurately calculate the health score of packages.

- **Adding markers** - Currently, the tool is calculating the health score based on 15 markers. Even though we are quite pleased with the results so far, the score would surely be more accurate if we could collect more markers. Therefore, in future work it would be worthwhile to consider adding more markers (and warnings) analysed in Chapter 4, such as number of packages the author owns, if he is reachable (active), community raising issues, etc.

- **Adding repositories** - Currently, the tool can provide health score for packages coming from 4 repositories; PyPi, npm, Maven and conda. The performance would significantly increase if more repositories would be added. Since Libraries.io collects the same metadata on packages coming from 28 different repositories, this should not impose a difficult implementation.

- **Further tests** - Currently, the tool is tested on approximately 200 packages which, in the scope of this research, we deemed as enough to conclude that the tool performs well. It would be worthwhile to perform further tests and get feedback on performance and any potential improvements.

Improve repository security
When we compared findings from Chapters 5 and 6, we came to the realization that higher repository security can already counter certain supply chain attacks. Maven's repository security is much better than the one of PyPi and npm, since it enforces strong namespace conventions, domain validation through Group ID, and a strong publish validation process. This already makes typosquatting, hostile takeover and dependency confusion attacks much more difficult, if not, almost impossible. That is why, we strongly recommend that repository maintainers adopt a robust namespace convention and two-factor authentication at a minimum, following the example set by Maven. To further help in combating supply chain attacks, there should be even more features in place. Repositories should increase transparency by providing more information about a package (author, maintainers, download count, etc.). Also, they should implement software composition analysis tools that would, similarly to our tool, provide a health score and alert users about old packages, outdated dependencies, potential vulnerabilities, etc. Furthermore, they should also have strong malware detection in place that would automatically block malicious packages from even being published. By implementing all of these features, developers would have access to all the critical information they need in one place, rather than having to rely on multiple tools. Therefore, we strongly believe that with increased security measures in place, supply chain attacks would be significantly more difficult to carry out, reducing the overall risk of such attacks.

Compensating maintainers of critical open-source projects
Paying developers who contribute significantly to critical (popular) open-source packages could help reduce the problem of abandoned or insufficiently maintained packages. Many popular packages are owned and maintained by individuals who work on them for free in their spare time, while companies rely on them to build applications that are usually monetized. By compensating these developers for their time and effort, it would provide a more sustainable model for maintaining these packages, and reduce the risk of them being neglected, containing vulnerabilities or even being abandoned. This approach would help ensure that open-source projects remain healthy and viable for the long-term, benefiting both individual contributors and companies alike.

9 Conclusions

Supply-chain attacks are rapidly growing and are here to stay, which is why we need to have adequate detection practices in place. In this thesis, we identified seven possible attack vectors of third-party dependencies: *1) Typosquatting, 2) Dependency confusion, 3) Vulnerable dependencies, 4) Abandonware, 5) Insufficiently maintained libraries, 6) Hostile takeover and 7) Transitive dependencies*, together with their markers, which can be found in Chapter 4. Certain categories have markers with higher accuracy (typosquating (1), dependency confusion (2), vulnerable dependencies(3)), while for others, markers have to be combined and placed in a context to provide higher assurance of a potential attack vector. Furthermore, we established that abandonware (4) is an extreme case of an insufficiently maintained library (5), and thus, their markers have to overlap.

It was quite challenging to provide exact thresholds for markers of those two categories, since they heavily depend on the type and size of the project in question. Also, thresholds can heavily depend on the individual perception and criticism of an open source project, so they can vary for different observers. For instance, for a certain user, two years with no activity or commits is a clear indicator that the project is abandoned, while for another, that threshold could be seen as too strict, especially if the project is small and does not require much upkeep. It is also important to highlight that transitive dependencies are a unique category, since they intersect with the other six categories and can depend on each of them.

The findings in Chapters 5 and 6 show that Nexus IQ, as utilized by Rabobank, is only capable of identifying vulnerable PyPi and Maven libraries; that contain known vulnerabilities (3). However, even though our simulation of the dependency confusion attack passed unnoticed, we discovered that Nexus Repository and Nexus Firewall do have features that can defend against it. Additionally, while it does not directly protect against typosquatting, using Nexus Repository and Firewall together should allow the detection and blocking of malware in typosquatted packages from entering the pipeline. It is also worth noting that both PyPi and Maven maintainers are quick to remove typosquatted packages, reducing the attack surface. However, Nexus IQ provides no protection against abandonware, insufficiently maintained libraries, and hostile takeover, which was the motivation behind developing a new tool (chapter 7). Finally, we concluded that Nexus IQ does analyze transitive dependencies, but can only detect vulnerable transitive dependencies. If combined with Nexus Repository and Firewall, it can also detect transitive dependency confusion and typosquatting.

Furthermore, there was a noticeable contrast between PyPi and Maven repositories, since Maven enforces strong namespace conventions. The name of every artifact consists of minimally three parts, where the most important is Group ID, since it is unique, and enforces the publisher to prove the ownership of the associated domain name. Consequently, this makes typosquatting and dependency confusion attacks extremely challenging to achieve on Maven packages, as evidenced by the lack of known examples. Maven also enforces a strong validation process when publishing a project or a new version, which requires an author's private key and a secret passphrase in addition to a password/email combination. This greatly reduces the risk of a hostile takeover attack through an expired domain hijack, making it close to impossible. This leads to the conclusion that improved security measures on the repository's side could effectively address most categories of supply chain attacks (1,2,6, 7).

The findings from Chapter 5 and Chapter 6 indicate that Rabobank's utilization of Nexus IQ only identifies vulnerable packages, while Firewall and Repository provide additional protection against packages that contain malware (such as typosquatting, dependency confusion, and hostile takeover). However, our analysis revealed that there is no protection against abandoned and insufficiently maintained packages. We also discovered that there is potential for implementing additional checks to identify hostile takeover and malicious packages, which are not overly complex to implement. As a result of these limitations, we were inspired to create our own tool

that can flag the categories that Nexus IQ is not capable of identifying. To be more specific, our tool is capable of calculating a health score for any open-source package found on PyPi, npm, Maven or conda. Additionally, it can also flag packages on certain characteristics and provide associated warnings to the user. These warnings can pinpoint abandoned packages, insufficiently maintained packages, hostile takeover through maintainer domain hijack (marker H_4, chapter 4), and certain malicious packages (typosquatting and dependency confusion if the package is very fresh or only has one version). If the score is higher than 75% with no warnings, the package is considered to be safe (sufficiently maintained). On the other hand, if a package receives a score lower than 50%, it is considered to be abandoned or insufficiently maintained. Packages with less than 60% score are suspected to be insufficiently maintained, and the label for the ones between 60% and 75% heavily depends on the context of warnings (if any). The score is calculated based on a formula inspired by Robin Pike [1], and 15 markers: *"created since, updated since, contributor count, organisation count, commit frequency, releases count, closed issues count, dependant projects count, dependencies updated, libraries.io rank, not deprecated, maintained, not removed, recent release and homepage accessible"*, which are collected from Libraries.io and GitHub API. There are two ways to use the tool; as a desktop application using GUI, or as a web application using API. After adjusting the weights to the "accurate" ones, we tested the tool on over 200 packages and were quite satisfied with the results since we deemed them as correct. It even correctly flagged certain popular packages as insufficiently maintained (manually validated), which was a positive surprise and further validation that it is indeed fulfilling its purpose, which is to detect and prevent possible supply chain attacks. Therefore, based on the results, we confidently claim that this tool performs very well and represents a valuable addition to Nexus IQ, in terms of protecting developers against supply chain attacks.

Finally, the comparison of repository security in PyPi, npm, and Maven revealed that stronger security measures, such as a robust namespace convention, two-factor authentication, and a strong publish validation process, would make supply chain attacks much more difficult to execute. To further combat these attacks, repositories should increase transparency, implement software composition analysis tools, and have strong malware detection in place. By implementing these measures, developers could access critical information in one place and reduce the risk of supply chain attacks. Furthermore, compensating developers who maintain critical (popular) open-source packages would provide a more sustainable model for maintaining these packages, and would reduce the risk of them being abandoned or neglected. This approach would help to ensure the long-term health and viability of open-source projects, benefiting both individual contributors and companies that rely on them. To conclude, enhancing repository security and compensating maintainers of essential projects could greatly diminish the risk of supply chain attacks.

References

[1] Criticality score. GitHub project. URL: https://github.com/ossf/criticality_score.

[2] GitHub REST API. GitHub Documentation. URL: https://www.sonatype.com/state-of-the-software-supply-chain/introduction.

[3] Httplib2 vulnerabilities. CVE. URL: https://www.opencve.io/cve?vendor=httplib2_project.

[4] Pytest-fork project readme. GitHub. URL: https://github.com/pytest-dev/pytest-forked#readme.

[5] Rollout plan for critical projects promo. GitHub. URL: https://github.com/pypi/warehouse/issues/11625.

[6] How to deprecate a pypi package. 2020. Blog. URL: https://www.dampfkraft.com/code/how-to-deprecate-a-pypi-package.html.

[7] sgmllib3k which is one of the dependencies, is deprecated. 2021. Github issues. URL: https://github.com/kurtmckee/feedparser/issues/279.

[8] API docs. 2022. Libraries.io Documentation. URL: https://libraries.io/api.

[9] Chaoss metrics. April 2022. Documentation. URL: https://chaoss.community/wp-content/uploads/2022/04/English-Release-2022-04-18.pdf.

[10] Cve details. 2022. Struts library. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-45/product_id-6117/version_id-394370/Apache-Struts-1.2.7.html.

[11] Cve details. 2022. Jackson-databind library. URL: https://www.cvedetails.com/vulnerability-list/vendor_id-15866/product_id-42991/Fasterxml-Jackson-databind.html.

[12] Source rank. 2022. Libraries.io Documentation. URL: https://docs.libraries.io/overview.html#sourcerank.

[13] Maven getting started guide. 2023. Documentation. URL: https://maven.apache.org/guides/getting-started/index.html#How_do_I_deploy_my_jar_in_my_remote_repository.

[14] Maven getting started guide. 2023. Documentation. URL: https://maven.apache.org/docs/3.8.1/release-notes.

[15] Juan Aguirre. Npm hijackers at it again: Popular 'coa' and 'rc' open source libraries taken over to spread malware. May 2021. Sonatype Blog. URL: https://blog.sonatype.com/npm-hijackers-at-it-again-popular-coa-and-rc-open-source-libraries-taken-over-to-spread-

[16] Fred Bals. What is a software bill of materials? March 2022. Synopsys Blog. URL: https://www.synopsys.com/blogs/software-security/software-bill-of-materials-bom/.

[17] J. Biden. Executive order on improving the nation's cybersecurity. 2021. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[18] Alex Birsan. Dependency confusion: How I hacked into Apple, Microsoft and dozens of other companies. *Medium, February*, 9, 2021.

[19] Cyber Safety Review Board. Review of the December 2021 Log4J Event. 2022. US department of Homeland Security. URL: https://www.cisa.gov/sites/default/files/publications/CSRB-Report-on-Log4-July-11-2022_508.pdf.

[20] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L. Silva. Is this github project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 122:106274, 2020.

[21] Jailton Coelho, Marco Tulio Valente, Luciana L. Silva, and Emad Shihab. Identifying unmaintained projects in github. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3239235.3240501 doi:10.1145/3239235.3240501.

[22] Federal Trade Commission. Equifax data breach settlement. February 2022. Official US Government website. URL: https://www.ftc.gov/enforcement/refunds/equifax-data-breach-settlement.

[23] Bleeping Computer. Pypi packages hijacked after developers fall for phishing emails. August, 2022. Blog. URL: https://www.bleepingcomputer.com/news/security/pypi-packages-hijacked-after-developers-fall-for-phishing-emails/.

[24] Josh Fruhlinger. Equifax data breach faq: What happened, who was affected, what was the impact? February 2020. Blog. URL: https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-was-affected-what-was-the-impact.html.

[25] Chris Good. New developer tools for open source dependency management, March 2022. Sonatype Blog. URL: https://blog.sonatype.com/new-developer-tools-for-open-source-dependency-management.

[26] Dan Goodin. How a college student tricked 17k coders into running his sketchy script. 2016. Arstechnica Online Journal. URL: https://arstechnica.com/information-technology/2016/06/college-student-schools-govs-and-mils-on-perils-of-arbitrary-code-execution/.

[27] Dan Goodin. Malicious npm packages are part of a malware "barrage" hitting repositories. 2021. Arstechnica Online Journal. URL: https://arstechnica.com/information-technology/2021/12/malicious-packages-sneaked-into-npm-repository-stole-discord-tokens/.

[28] Kadi Grigg. Considering nexus auditor? you should, but know these things first. 2020. Sonatype Blog. URL: https://blog.sonatype.com/considering-nexus-auditor-you-should-but-know-these-things-first.

[29] Heavy.AI. Open source library. Blog. URL: https://www.heavy.ai/technical-glossary/open-source-library#:~:text=An%20open%20source%20library%20is,and%2For%20publish%20without%20permission.

[30] Anton Hoffman. Solarwinds orion security breach: A shift in the software supply chain paradigm. 2021. Blog. URL: https://snyk.io/blog/solarwinds-orion-security-breach-a-shift-in-the-software-supply-chain-paradigm/.

[31] ICANN. EPP Status Codes, What Do They Mean, and Why Should I Know? URL: https://www.icann.org/resources/pages/epp-status-codes-2014-06-16-en.

[32] Louis Lang. Malicious python packages replace crypto addresses in developer clipboards. November, 2022. Blog. URL: https://blog.phylum.io/pypi-malware-replaces-crypto-addresses-in-developers-clipboard.

[33] Libraries.io. Pytest-fork library information. Pypi package statistics. URL: https://libraries.io/pypi/pytest-forked.

[34] npm Docs. Deprecating and undeprecating packages or package versions. Documentation. URL: https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-package-versions.

[35] npm Docs. npm-fund. Documentation. URL: https://docs.npmjs.com/cli/v7/commands/npm-fund.

[36] Michael Prescott. Namespace confusion: Minimizing risk with nexus repository. February, 2021. Sonatype Blog. URL: https://blog.sonatype.com/namespace-confusion-minimizing-risk-with-nexus-repository.

[37] Rope Security. Vulnerable dependencies. Article. URL: https://ropesec.com/articles/vulnerable-dependencies/.

[38] Nikita Skovoroda. Gathering weak npm credentials. July 2017. Research article. URL: https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md.

[39] Sonatype. Nexus auditor. Official website. URL: https://www.sonatype.com/products/auditor.

[40] Sonatype. Nexus container. Official website. URL: https://www.sonatype.com/products/container.

[41] Sonatype. Nexus firewall. Official website. URL: https://www.sonatype.com/products/firewall#:~:text=Nexus%20Firewall%20is%20the%20only,Automatic%20protection%20from%20unknown%20risks.

[42] Sonatype. Nexus lifecycle. Official website. URL: https://www.sonatype.com/products/open-source-security-dependency-management.

[43] Sonatype. Nexus repository. Official website. URL: https://www.sonatype.com/products/nexus-repository.

[44] Sonatype. Requirements. Sonatype Documentation. URL: https://central.sonatype.org/publish/requirements/.

[45] Sonatype. Sonatype documentation, licensing and features. Official website. URL: https://help.sonatype.com/iqserver/product-information/licensing-and-features.

[46] Sonatype. Sonatype guides, understanding vulnerability data. Official Documentation. URL: https://guides.sonatype.com/iqserver/technical-guides/sonatype-vuln-data/#how-does-sonatype-provide-high-quality-data.

[47] Sonatype. Sonatype stops software supply chain attack aimed at the java developer community. Sonatype Blog. URL: https://blog.sonatype.com/malware-removed-from-maven-central.

[48] Sonatype. What are "sonatype" vulnerability ids? Sonatype Documentation. URL: https://support.sonatype.com/hc/en-us/articles/4406975343123-What-are-Sonatype-vulnerability-IDs-.

[49] Sonatype. Breaking Down Biden's Cybersecurity Executive Order. 2021. Blog. URL: https://www.sonatype.com/breaking-down-bidens-cybersecurity-executive-order.

[50] Sonatype. State of the software supply chain. 2022. URL: https://www.sonatype.com/state-of-the-software-supply-chain/introduction.

[51] Christian Stein. Github project "modules". 2022. URL: https://github.com/sormuras/modules.

[52] Synopsys. Open source security and risk analysis report. 2022. Technical Report. URL: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf.

[53] Fossa Editorial team, Jun 2022. Blog. URL: https://fossa.com/blog/dependency-confusion-understanding-preventing-attacks/#:~:text=Dependency%20confusion%20is%20a%20software,a%20package%20before%20private%20registries.

[54] Cybersecurity news The Daily Swig and views. Malicious python library ctx removed from pypi repo. May, 2022. Cyber security news and views provided by Port swigger Web Security. URL: https://portswigger.net/daily-swig/malicious-python-library-ctx-removed-from-pypi-repo.

[55] Read the Docs Team. Policy for abandoned projects. Documentation. URL: https://docs.readthedocs.io/en/stable/abandoned-projects.html.

[56] Tidelift. Up to 20 percent of your application dependencies may be unmaintained. Company blog. URL: https://blog.tidelift.com/up-to-20-percent-of-your-application-dependencies-may-be-unmaintained.

[57] Hugo van Kemenade. Json file with download statistics for top PyPi packages per month. October, 2022. URL: https://hugovk.github.io/top-pypi-packages/top-pypi-packages-30-days.min.json.

[58] Owen Williams. The internet relies on people working for free. September 2019. Blog. URL: https://onezero.medium.com/the-internet-relies-on-people-working-for-free-a79104a68bcc.

[59] Daniel Wisenhoff. Vulnerabilities in dependencies, third party components and open source: What you need to know. February 2021. Blog. URL: https://debricked.com/blog/vulnerabilities-dependencies/.

[60] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 331–340. IEEE, 2022.

[61] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.