Engineering Degree Project

Supplementing Dependabot's vulnerability scanning

A Custom Pipeline for Tracing Dependency Usage in JavaScript Projects

Authors:	Isak Karlsson
	David Ljungberg
Supervisors:	Andreas Johansson
	Jesper Johansson
LNU Supervisor:	Fredrik Ahlgren
Semester:	Spring 2023
Subject:	Computer Science

Abstract

Software systems are becoming increasingly complex, with developers frequently utilizing numerous dependencies. In this landscape, accurate tracking and understanding of dependencies within JavaScript and TypeScript codebases are vital for maintaining software security and quality. However, there exists a gap in how existing vulnerability scanning tools, such as Dependabot, convey information about the usage of these dependencies. This study addresses the problem of providing a more comprehensive dependency usage overview, a topic critical to aiding developers in securing their software systems. To bridge this gap, a custom pipeline was implemented to supplement Dependabot, extracting the dependencies identified as vulnerable and providing specific information about their usage within a repository. The results highlight the pros and cons of this approach, showing an improvement in the understanding of dependency usage. The effort opens a pathway towards more secure software systems.

Keywords: Vulnerability scanning, Software Dependencies, JavaScript, TypeScript, Dependabot, Vulnerability Scanning Tools, Software Security, Pipeline, GitHub, Dependency Management

Contents

1 Introduction

In this digital age, the interconnection of various sectors of modern society through software systems has rapidly evolved, shaping our way of life and streamlining the flow of information. We heavily rely on these systems in virtually every aspect of life, from healthcare and finance to communication and transportation [1]. They facilitate everything from the instantaneous transfer of funds globally to the tracking of health records, and from enabling instant communication to ensuring the efficient operation of our transportation systems. However, as society becomes more digitized and interconnected, and as our dependence on these software systems increases, so does the potential for their exploitation by cybercriminals.

1.1 Background

Software vulnerabilities are weaknesses in a software system's design, implementation, or configuration that can be exploited by an attacker to breach the system's confidentiality, integrity, or availability. The digitalization of our lives and the widespread connectivity between systems today means that these vulnerabilities could have far-reaching consequences. Financial losses can be significant, with companies suffering from data breaches often facing hefty fines, litigation costs, and the expenses of implementing new security measures. Reputational damage can also be severe, as public trust in companies is hard to regain after a breach. More critically, when vulnerabilities affect systems integral to human life, such as healthcare systems or transportation infrastructure, the risks extend beyond financial and reputational damage. They may result in endangerment of human lives, demonstrating the paramount importance of securing our digital society [2].

In contemporary software development, the widespread use of third-party libraries and software dependencies adds a layer of complexity to the task of securing software systems. Dependencies are external software components on which a software project relies for its proper functionality. While they accelerate the development process and reduce code duplication, they also introduce potential vulnerabilities that come from these external components, necessitating the need for robust solutions for identifying and mitigating these vulnerabilities [3, 4].

The shift towards more dependency-heavy software development practices, combined with the heightened interconnectivity of the digital world, underscores the urgent need for effective vulnerability scanning. Vulnerability scanning involves the proactive analysis of these dependencies to identify potential weaknesses that attackers could exploit. By staying informed about their dependencies' security risks and patching or updating them as necessary, developers can minimize their software system's attack surface.

The importance of vulnerability scanning in software dependencies has been further accentuated by high-profile security breaches and vulnerabilities, such as the Equifax data breach in 2017 and the Log4Shell vulnerability discovered in 2021. The Equifax breach, which compromised the personal information of over 145 million individuals, was traced back to a vulnerability in the Apache Struts web application framework—a third-party dependency used by Equifax [5]. On the other hand, the Log4Shell vulnerability, found in the widely used Log4j logging library, allowed remote code execution by attackers and affected countless software systems across various industries [2]. These incidents, among others, underscore the critical need for comprehensive vulnerability scanning in software dependencies to protect sensitive data, ensure the continued functionality of software systems, and maintain public trust in digital infrastructure.

Given the increasing volume of dependencies and the continual discovery of new vulnerabilities, the task of manual analysis and patching becomes impractical and time-consuming. Automated scanning tools have emerged as indispensable for identifying vulnerabilities in software dependencies quickly and accurately. Using techniques like static and dynamic code analysis, dependency checking, and software composition analysis, these tools can scan for known vulnerabilities, prioritize the most critical issues, and reduce the workload of security teams [6, 7].

Several open-source and commercial automated scanning tools are available in the market, each offering different features, capabilities, and limitations [6, 8]. For developers and organizations, choosing the right tool is crucial for maximizing the effectiveness of vulnerability scanning and ensuring the security of their software systems [7]. In light of these considerations, Dependabot was selected for this project given its feature set and integration with the GitHub ecosystem [9].

This degree project revolves around developing a custom pipeline that supplements Dependabot. The pipeline utilizes Dependabot's findings, particularly the vulnerabilities it identifies in software dependencies, and enriches this information by indicating where these dependencies are used within a repository. This process is aimed at providing insights to aid developers and organizations in building more secure and efficient software ecosystems in our ever-increasing digital world.

In conclusion, this degree project will contribute to understanding and mitigating vulnerabilities in software dependencies, a crucial endeavor in safeguarding the integrity, confidentiality, and availability of software systems in our increasingly interconnected and digitally dependent society.

1.2 Related work

Current literature has explored the features and effectiveness of open-source vulnerability scanning tools and assessed the practicality of dependency management bots, such as Dependabot. However, a gap exists in how these tools, including Dependabot, communicate information related to the usage of vulnerable dependencies. This is particularly vital for developers in understanding how and where vulnerabilities might impact their projects if updating the dependencies fails to resolve security issues.

In one study, titled "Do developers update their library dependencies?" [10], the authors investigate the extent to which developers update their library dependencies. They conducted an empirical study on library migration that covers over 4,600 GitHub software projects and 2,700 library dependencies. Results show that 81.5% of the studied systems still keep their outdated dependencies, and 69% of the developers were unaware of their vulnerable dependencies. This study underscores the importance of automated tools like Dependabot in managing software vulnerabilities and demonstrates the need for more effective communication of vulnerability information to developers.

In another notable study, titled "Analysis of Open Source Node.js Vulnerability Scanners" [6], the authors compare open-source vulnerability scanning tools Retire.js and Snyk. They conclude that Snyk offers more features and a larger vulnerability database, while Retire.js is recommended by OWASP (Open Web Application Security Project) for scanning Node and JavaScript vulnerabilities.

Another important study, "Automating Dependency Updates in Practice: An Exploratory Study on GitHub Dependabot" [8], examines the effectiveness of the dependency management bot Dependabot in updating software dependencies and reducing notification fatigue. The findings are mixed, and the authors suggest that an ideal bot should have configurability, autonomy, transparency, and self-adaptability. Despite providing numerous features and capabilities for vulnerability scanning and dependency management, existing tools currently lack the ability to automatically identify whether the code uses a vulnerable part of the dependency.

In the report "Vulnerable Open Source Dependencies: Counting Those That Matter" [11], the authors propose a precise methodology for analyzing vulnerable dependencies in open-source software (OSS) libraries. By combining code-based analysis with build, test, update dates, and group information from code repositories, the approach aims to improve resource allocation for development and auditing. A case study of 200 popular OSS Java libraries used by the company SAP revealed that 20% of known vulnerable dependencies were not deployed and thus not exploitable. The study also found that developers were able to fix 82% of deployed vulnerable dependencies, with 81% of them resolved by updating to a new version, while 1% required more costly mitigation strategies. This methodology provides actionable information for software development companies to efficiently allocate resources for managing library dependencies.

In the study "Bots Don't Mind Waiting, Do They? [12] Comparing the Interaction With Automatically and Manually Created Pull Requests" a GitHub mining study was conducted to identify measurable differences in how maintainers interact with pull requests created manually by humans versus those automatically generated by bots. The study found that about one-third of all pull requests on GitHub currently come from bots. Yet, while pull requests from humans were accepted and merged in 72.53% of all cases, only 37.38% of bot-generated pull requests were treated the same. Moreover, bot-generated pull requests took significantly longer to be interacted with and merged, despite containing fewer changes on average than human-generated pull requests. These findings imply that the potential of bots in software development is still under-realized. Additionally, the study revealed that Dependabot was the most frequently used bot, having created as many as 3,000,000 pull requests and the second most bot was dependabot-preview with 1,200,000. This demonstrates Dependabot's widespread usage within the GitHub community, highlighting its importance in the landscape of software development tools. These findings underscore the need for further research into how bots can be better integrated into software development workflows and how their contributions can be more effectively evaluated and utilized [12].

1.3 Problem formulation

An empirical study on library migration reveals that while many systems heavily rely on library dependencies, a staggering 81.5% continue to use outdated dependencies. Furthermore, when these outdated dependencies are identified as having vulnerabilities, 69% of developers do not update their code to resolve these emerging problems [10].

The current degree project addresses this issue by implementing a custom pipeline that leverages GitHub's Dependabot. Dependabot alerts developers to these vulnerabilities, by creating "Dependabot Alerts", and creates pull requests to update dependencies where patched versions are available.

The custom pipeline implemented for this project supplements the functionality of Dependabot by providing additional information on the exact usage locations of these dependencies within the codebase. This is accomplished by editing the descriptions of pull requests and creating issues. Considering that bot-generated pull requests currently have a 37.38% acceptance rate [12]. The additional context provided by this custom pipeline could potentially increase the likelihood of acceptance by developers, thereby improving the security of the project.

RQ1 How can the custom pipeline supplement Dependabot by integrating dependency usage information and how can it be communicated to developers?

RQ2 How efficacious is the custom solution in identifying the actual usage of dependencies, and to what extent does it inaccurately report false positives or false negatives?

1.4 Motivation

The motivation for this degree project lies at the intersection of modern software development practices and the dire need to secure software systems. For companies, securing software systems against vulnerabilities is not just a technical concern but a crucial business strategy. Any exploitation of these vulnerabilities can result in significant financial losses, reputational damage, and regulatory penalties, all of which could jeopardize business continuity. Moreover, in a competitive business landscape, consumers demand that their sensitive data be protected [2, 4]. Any compromise in data security could lead to loss of customer trust and business opportunities. By providing developers with a deeper understanding of where vulnerabilities exist and their precise usage within the codebase, the custom pipeline implemented in this project can help businesses mitigate the risks associated with software dependencies. Additionally, by enriching Dependabot's output with additional information, the pipeline could potentially speed up the resolution process by equipping developers with more context. This increased efficiency could reduce the time during which systems remain vulnerable, thereby enhancing overall security posture and, in turn, customer trust.

For the broader society, the stakes are even higher. As we come to rely on software systems for critical infrastructure, from healthcare and finance to transportation and utilities, ensuring the integrity of these systems becomes a matter of public safety and social stability [2, 4]. By bolstering our ability to identify and remediate software vulnerabilities, we contribute to the broader endeavor of protecting our digital society against cyber threats.

Furthermore, this degree project is aligned with global efforts to improve software security practices and foster a culture of shared responsibility in securing digital infrastructure. Through this research, it is aimed not only to enhance the technical tools but also to contribute to the understanding of how improved vulnerability management practices can be implemented and propagated across the software development community.

1.5 Scope/Limitation

The primary focus is on software vulnerability management and dependency management tools, with special emphasis on JavaScript-based projects. The discourse is mainly situated within the context of GitHub and its tool Dependabot. Furthermore, it investigates the effectiveness of a custom solution in delivering information to developers to aid in dependency management and software security. The implementation and evaluation of this custom solution form a substantial part of the study, aiming to uncover its strengths and weaknesses and its overall impact on software security.

However, it is important to note certain limitations of this study. While the thesis focuses on JavaScript and TypeScript projects, the findings may not be entirely applicable to projects based on other programming languages due to differences in ecosystems, libraries, and best practices. Also, while GitHub and Dependabot are widely used, there are other platforms and tools in use that this thesis does not cover.

Moreover, the evaluation of this custom solution is based on a selected set of repositories, which might not encompass all possible scenarios or issues faced in real-world software development environments. Lastly, while the thesis aims to provide an in-depth understanding of the covered topics, it is not exhaustive and does not claim to cover all aspects of software vulnerability and dependency management. Future research could further expand on the areas touched upon in this thesis.

1.6 Target group

The target group for this research includes software developers, security professionals, and organizations that rely on JavaScript and TypeScript projects. Additionally, this research may be of interest to developers and organizations using other programming languages and platforms, as the principles and techniques discussed can be adapted to different contexts. By improving dependency management and vulnerability scanning practices, these target groups can enhance the security and reliability of their software projects, benefiting both developers and end-users.

1.7 Outline

This thesis explores critical aspects of software development: vulnerability management and dependency management tools. The document is structured into several chapters. The introduction outlines the background, related work, the problem statement, its significance, the projected achievements, the scope and limitations, and the target audience, while providing a brief guide to the thesis contents.

The theory chapter clarifies essential concepts associated with software vulnerabilities such as the Common Vulnerability Scoring System, Common Vulnerabilities and Exposures, and Common Weakness Enumeration, and introduces GitHub and its tool Dependabot. This is followed by a detailed presentation of the study in the chapter method. this including the research project description, a literature review, the development of the implementation, a controlled experiment, and an evaluation of reliability, validity, and ethical considerations.

In the implementation chapter, the developed pipeline created for this research is described, leading into the experimental setup and results, where the conducted experiments and their outcomes are shared. This feeds into the analysis chapter, which delves into the experimental results, drawing conclusions based on the collected data.

The findings and the rationale behind the choices made throughout the project are then discussed. The thesis concludes with a summary of key findings and recommendations for future research in the field of software vulnerability and dependency management, aiming to provide a comprehensive overview and suggest potential next steps for further exploration in the field.

2 Theory

This chapter provides an overview of the fundamental concepts and related topics that form the foundation of this degree project. These topics are essential for understanding the context, challenges, and solutions associated with automated scanning for vulnerabilities in software dependencies.

2.1 Standardized Vulnerability Assessment Tools

Three main standardized tools, namely the Common Vulnerabilities Exposures (CVE), Common Vulnerability Scoring System (CVSS), and Common Weakness Enumeration (CWE), can play a significant role in the identification and assessment of software vulnerabilities. These tools provide a consistent and comprehensive means of understanding and managing the software vulnerabilities.

The CVSS is a tool used to measure the severity of software vulnerabilities. It applies three groups of metrics - Base, Temporal, and Environmental - that collectively create a score representing the seriousness of a given vulnerability. This scoring mechanism is fundamental in assessing the severity of vulnerabilities, thereby guiding the prioritization and mitigation strategies [13, 14]. CVSS scores are primarily provided by the National Vulnerability Database (NVD) and can be calculated using the CVSS calculator or implementing the CVSS equations specified in the CVSS version 3.1 specification document [15].

The CVE program serves as a comprehensive dictionary of identified vulnerabilities for specific codebases, providing unique identifiers, CVE IDs, for each vulnerability [16]. The CVE program is maintained by the MITRE Corporation, an American non-profit organization, sponsored by the U.S. Department of Homeland Security (DHS) and the Cybersecurity and Infrastructure Security Agency (CISA), and is a critical resource for researchers and cybersecurity personnel alike. Upon publication to the CVE list, each CVE entry is analyzed by the NVD to associate Reference Tags, CVSS scores, Common Weakness Enumeration (CWE) entries, and Common Platform Enumeration (CPE) Applicability statements [17, 18].

The CWE is a community-developed list of software and hardware weaknesses with security implications, providing a standardized taxonomy for classifying these weaknesses. Originating as an extension of the CVE project to provide a more granular classification system for code security assessment, the CWE list and classification taxonomy launched in 2006 offer comprehensive definitions and descriptions of prevalent weaknesses [19].

The connection between these systems can be illustrated using the well-known vulnerability, Log4Shell (CVE-2021-44228) [20]. The Log4Shell vulnerability is an exploit in the popular Java library log4j that allows remote code execution by an attacker. CVE provides the unique identifier for the Log4Shell vulnerability (CVE-2021-44228), making it easily trackable and referenceable. The associated CVSS score, given as 10.0, signifies the highest possible severity due to its potential for remote code execution with no user interaction required and low complexity for exploitation. The CWE reference (CWE-917), categorizes this vulnerability as an "Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection')," giving context to the type of weakness exploited [21]. Thus, CVE offers the unique identification, CVSS gives a quantifiable measure of the severity, and CWE provides a classification of the weakness exploited. This joint information helps developers, security researchers, and IT professionals to assess, prioritize, and remediate vulnerabilities such as Log4Shell effectively.

2.2 GitHub

GitHub is a widely used web-based platform for version control and collaboration that allows developers to work together on projects. Built on top of Git, a distributed version control system, GitHub provides a user-friendly interface and a set of features that facilitate software development, issue tracking, and team collaboration [22].

GitHub Actions is a feature provided by GitHub that enables developers to create custom, automated workflows called pipelines. These pipelines can be triggered by various events, such as pushing code to a repository, creating a pull request, or merging changes. Pipelines typically consist of a series of steps, known as actions, which can be used to build, test, and deploy code, among other tasks. GitHub Actions simplifies the process of setting up and maintaining continuous integration (CI) and continuous deployment (CD) pipelines. With a marketplace of community-contributed actions, developers can automate common tasks, such as running tests, building artifacts, and deploying applications to various environments [23].

The GitHub API is an interface that allows developers to interact with GitHub programmatically. With the API, developers can perform various tasks, such as managing repositories, branches, and issues, as well as collaborating with other users on GitHub. The API is based on REST principles and uses standard HTTP methods (GET, POST, PUT, DELETE) for communication. Developers can access the API using their preferred programming language or tools, making it possible to integrate GitHub functionality into custom applications or scripts. The API also provides web-hooks for real-time notifications of specific events, enabling developers to build custom integrations and automate processes based on GitHub activity [24].

GitHub along with its pipelines and API, offers a set of tools that support software development, collaboration, and automation. These features can be relevant in the context of managing software dependencies and implementing vulnerability scanning and remediation workflows.

2.3 Dependabot

Dependabot is a dependency management tool that automates the process of updating project dependencies in software development projects hosted on GitHub [25]. Initially developed as an independent tool, Dependabot was acquired by GitHub in 2019 and is now fully integrated into the platform [8]. Its primary goal is to help developers maintain up-to-date and secure project dependencies by automating the process of finding, reviewing, and applying updates [26]. This service encompasses two major functionalities: Automated Dependency Updates and Security Updates. Automated Dependency Updates is a proactive service triggered by a repository configuration file, facilitating the creation of pull requests for dependency updates. On the other hand, Security Updates involve a comprehensive scan of GitHub repositories to identify potential security vulnerabilities in dependencies, notifying repository owners of identified issues and their severity. The latter also includes Dependabot Vulnerability Alerts, an integral part of Security Updates that provide repository owners with vital vulnerability-related information [27, 28].

2.3.1 Automated Dependency Updates

Automated Dependency Updates are initiated when a configuration file named dependabot.yml is added to a GitHub repository. Dependabot then starts opening pull requests to update project dependencies to their latest versions. These updates help developers maintain up-to-date dependencies, reducing the risk of compatibility issues and improving overall software stability.

Developers can customize Dependabot's behavior by specifying various options in the dependabot.yml file, such as which dependencies to update, the update interval, and the maximum number of simultaneous Dependabot pull requests [29]. This allows developers to prioritize specific dependencies, control the frequency of updates, and manage the workload associated with reviewing and merging pull requests.

2.3.2 Security Updates

Security Updates involve Dependabot continuously scanning the entire GitHub platform to identify repositories using dependencies with known security vulnerabilities. This proactive approach helps developers address security issues before they can be exploited, reducing the likelihood of successful cyberattacks on their projects.

Even without a dependabot.yml file, Dependabot alerts repository owners about vulnerable dependencies via email notifications, providing them with information about the severity and potential impact of the vulnerability. Repository owners can then manually instruct Dependabot to open Pull Requests that update the vulnerable dependencies to their patched versions [27]. These security-focused updates help developers maintain a secure codebase, protecting their projects and users from potential harm caused by exploited vulnerabilities.

The GitHub Advisory Database is a resource that consolidates information on known security vulnerabilities from various sources, including the industry-standard CVE database. Incorporating CVE data into the GitHub Advisory Database ensures comprehensive coverage of vulnerabilities across different software components and platforms [30, 31].

Many vulnerabilities listed in the GitHub Advisory Database are also scored using CVSS, a standard for assessing the severity of security vulnerabilities. It is important to note that not all advisories in the GitHub Advisory Database have a CVSS score, particularly if they are not part of the CVE database or if they are newly disclosed vulnerabilities that have not yet been scored [32]. Dependabot utilizes the information available in the GitHub Advisory Database to identify and address security vulnerabilities in software dependencies [30, 31].

In addition to the CVE and CVSS scoring system, the GitHub Advisory Database also incorporates information from the CWE system. The CWE helps developers address the underlying issue rather than merely applying a patch or workaround. This allows for more effective and long-lasting security improvements in the software development process. Dependabot can leverage CWE information in conjunction with the GitHub Advisory Database to provide developers with a more comprehensive understanding of the vulnerabilities in their software dependencies. This enables them to make informed decisions about patching, updating, removing, or replacing dependencies to mitigate security risks and improve the overall security posture of their software systems [30, 31].

2.3.3 Dependabot vulnerability Alerts

Dependabot Vulnerability Alerts is part of the Security Updates offered by Dependabot. They are generated when Dependabot identifies a repository using dependencies with known security vulnerabilities. These alerts provide essential information to repository owners, such as details about the vulnerability, its severity based on CVSS scores, and potential impacts [27, 28].

Dependabot sends these alerts to repository owners via email notifications. These alerts inform developers about vulnerable dependencies in their projects and also provide them with an option to update the vulnerable dependency directly from the alert interface if a patch is available [27, 28].

Dependabot Vulnerability Alerts make use of the data available in the GitHub Advisory Database, which includes information from the CVE database, CVSS scores, and CWE data [27].

3 Method

A multi-step method was used in this study. Firstly, literature on dependency management tools was reviewed. Subsequently, the challenges associated with dependency management were explored. Lastly, an implementation was developed and evaluated, it was specifically designed to detect dependency imports and usages in JavaScript and TypeScript projects. The subsequent sections detail the steps undertaken during this process, which include a literature review, the development of the implementation, and a controlled experiment. The specifics of this implementation will be discussed further in Chapter 4.

3.1 Research Project

A series of different stages, as depicted in Fig. 3.1, were followed in this research project. A variety of methods were applied to address each phase of the project.

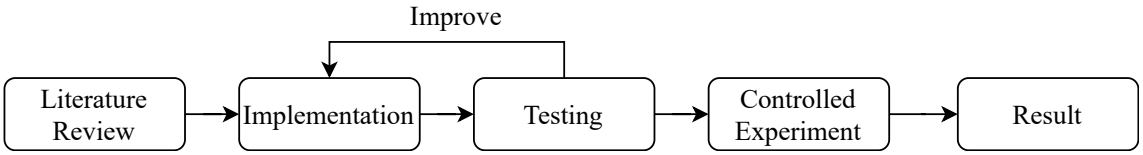

Figure 3.1: Flowchart of the controlled experiment process.

The process begins with a review of existing literature on automated dependency management, vulnerability scanning tools, and package managers. This approach is chosen as it facilitates a comprehensive understanding of the subject. The literature review establishes a robust foundation for the project and the implementation phase.

Following the literature review, the implementation is constructed and tested against various JavaScript projects. This stage enables iterative refinement of the implementation, enhancing its performance and effectiveness.

Once the implementation reaches a satisfactory level of performance, the controlled experiment method is applied. This technique involves designing and conducting an experiment to evaluate the efficiency of the implementation in detecting imports and the usage of dependencies. The insights obtained from these experiments guide further refinement of the implementation, ensuring it meets the desired standards of accuracy and reliability.

In conclusion, the methodical approach adopted in this research project is designed to ensure a thorough and accurate evaluation of the implementation's efficacy in detecting dependency imports and usage. It involves a systematic progression from literature review to implementation, testing, and final evaluation. This multi-phase process aims to maximize the reliability of the findings and allows for continuous improvement and refinement of the implementation. The ultimate goal is to deliver a solution that effectively aids developers in managing dependencies and addressing potential vulnerabilities, thereby enhancing the security and stability of JavaScript and TypeScript projects.

3.2 Literature Review

The literature review process for this project focused on automated dependency management, vulnerability scanning tools, and package managers. Using platforms such as Elicit.org, Google Scholar, and Linnaeus University Library's OneSearch, it sought to understand the current state-of-the-art in vulnerability scanning tools and identify areas for further exploration.

Elicit.org is an AI-powered search engine designed to assist researchers in finding relevant academic papers. It employs a combination of machine learning techniques and natural language processing (NLP) to analyze the content of scholarly articles, thereby generating search results that are both relevant and comprehensive. Elicit searches through a corpus of 115M papers from the Semantic Scholar Academic dataset. The platform continually updates and refines its results, based on AI-driven insights and user interactions, which makes the number of search results highly dynamic, adaptable, although impossible to accurately display in the table 3.2 due to the constant AI-driven updates [33].

The search strategy aimed to assess the relevance, project alignment, and publication date of paper. Following an initial screening, each abstract was read for deeper relevance, potentially leading to a comprehensive review. This approach, balancing breadth and focus, laid the groundwork for the research. To limit the number of papers, a set of inclusion/exclusion criteria from Table 3.2 was applied. Elicit's AI-driven search capabilities were utilized to uncover pertinent studies that traditional search engines might have missed. Additionally, references from the identified papers were examined to ensure a more comprehensive coverage of the research landscape. The goal was to read not more than a total of 20 papers, prioritizing those meeting the specified criteria.

Table 3.1: Inclusion and exclusion criteria for literature review.

Inclusion Criteria	**Exclusion Criteria**
Studies comparing different vulnerability scanning tools and automated patching tools specifically for JavaScript and TypeScript projects.	Studies that focus solely on theoretical aspects without any empirical evidence or practical application.
Research published in peer-reviewed journals, conference proceedings, or technical reports to ensure credibility and scientific rigor.	Studies with a primary focus on programming languages other than JavaScript and TypeScript, as they fall outside the scope of this project.
Research that provides a clear and robust methodology for assessing the effectiveness of the tools.	Studies focusing on outdated, discontinued, or rarely used tools with minimal relevance to current development practices.
Studies focusing on open-source or commercially available tools. Emphasis is placed on tools that are actively maintained and widely used.	Research published over a decade ago, as the tools and techniques discussed may be obsolete or irrelevant in the context of contemporary projects.
Studies that provide insights into practical applications or real-world case studies of the tools and techniques in question.	Studies that only peripherally mention the topics of interest without providing in-depth analysis or relevant findings.
Research that includes statistical analysis or quantitative data to support findings.	Studies that do not provide information on their data collection.
Research that includes user feedback, developer experiences, or user studies related to the tools and techniques being investigated.	Studies that focus on proprietary tools or systems that are not accessible or applicable to the wider development community.

Table 3.2 presents search terms related to dependency management, focusing specifically on vulnerabilities in third-party and open-source code.

Table 3.2: Number of studies found in different search tools for specified search terms.

Search term	OneSearch	Google Scholar	Elicit
Third-party dependencies	706	656 000	N/A
Open source dependencies	3 203	2 400 000	N/A
Updating dependencies	797	500 000	N/A
Automated dependency updates	10	61 000	N/A
Dependency management tools	1	883 000	N/A

3.3 Implementation Development

The implementation development phase focuses on constructing and refining the system that identifies dependency imports and usages in JavaScript and TypeScript projects. This stage is instrumental in bringing to life the concepts and strategies uncovered during the Literature Review.

The implementation employs GitHub's Dependabot, an automated tool designed to scan GitHub repositories for outdated or vulnerable dependencies and alert developers that they are using these dependencies. This implementation enhances Dependabot's functionality by appending information about where the dependencies are actually used in the codebase, editing pull request descriptions, and creating issues.

The development process involves iterative cycles of coding, testing, and debugging to enhance the system's functionality and reliability. The performance and accuracy of the implementation are continuously monitored and adjusted to maximize the detection of true dependency imports and usages while minimizing false positives and negatives.

Upon reaching a satisfactory level of performance, the implementation is then subjected to the Controlled Experiment phase, where its effectiveness is evaluated and tested on a new repository.

3.4 Controlled Experiment

Following the implementation phase, a controlled experiment is conducted to evaluate the effectiveness of the implementation in identifying dependency imports and usages within JavaScript and TypeScript projects.

A controlled experiment allows for the systematic assessment of the implementation under predetermined conditions, ensuring that the results obtained are not influenced by extraneous variables. This method is particularly suited to this study as it provides a structured approach to test the hypotheses in a real-world setting, allowing for reliable and valid results.

The controlled experiment involves applying the implementation to a selected repository that uses JavaScript and TyperScript. Once the implementation is applied to the repository, the results are collected and analyzed. The primary metrics of interest are the rates of true positives, false positives, and false negatives in identifying dependency imports and usages. These metrics provide direct insight into the implementation's accuracy and reliability.

Finally, the findings of the controlled experiment are documented, providing a comprehensive assessment of the implementation performance in detecting dependency imports and usages. These results not only validate the implementation but also contribute valuable insights to the field of automated dependency management and vulnerability scanning.

3.4.1 Evaluation Metrics

The performance of the custom implementation is evaluated through a combination of metrics that provide unique insights into its capabilities. True Positives (TP) denote the number of instances where the implementation accurately identifies a dependency in the codebase. False Positives (FP) are instances where the implementation incorrectly identifies a dependency, with a lower count reflecting accuracy. False Negatives (FN) are instances where the implementation fails to detect a dependency present in the code, a lower FN count suggests comprehensive detection.

Beyond these individual metrics, the overall performance of the implementation is quantified through a set of composite measures. Precision is calculated in Eq. 1 as the ratio of TP to the sum of TP and FP, providing a measure of the likelihood that a detected dependency usage is correct.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

Recall calculated in Eq. 2, is the ratio of TP to the sum of TP and FN. It indicates the probability of the implementation identifying all instances of a dependency.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

To provide a comprehensive assessment of the implementation's performance in a single number, the F_1-score is calculated in Eq. 3. It is the harmonic mean of Precision and Recall, thus providing a balanced measure that takes into account both false positives and false negatives.

$$F_1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

These evaluation metrics collectively provide an assessment of the implementation's performance, taking into account aspects such as accuracy and coverage. Evaluating the bot's performance using these metrics is crucial to ensuring it meets desired performance standards and provides valuable insights to developers regarding dependency usage and software security.

3.5 Reliability and Validity

Reliability refers to the consistency and repeatability of the research findings. To enhance the reliability of the study, the following measures have been adopted: conducting a literature review, designing and conducting a controlled experiment, and providing comprehensive documentation. By conducting a structured and thorough literature review, it is ensured that the information gathered is representative of the current state of knowledge in the domain of dependency management and vulnerability scanning. The controlled experiment is designed to minimize biases and potential confounding factors that could affect the results. Comprehensive documentation of the research project, including the design, implementation, and evaluation, helps to maintain transparency and facilitates the reproducibility of the study.

Validity refers to the accuracy and truthfulness of the research findings. To enhance the validity of the study, the following measures have been adopted: selecting appropriate research methods, using appropriate evaluation metrics, and acknowledging and addressing limitations and assumptions. By selecting suitable research methods, such as a literature review and controlled experiment, the research objectives are addressed effectively, and the findings are relevant and meaningful. The use of appropriate evaluation metrics, such as precision, recall, and F_1-score, helps to assess the performance of the created implementation in a meaningful and comparable manner. By acknowledging and addressing the limitations and assumptions of the study, the aim is to provide a more accurate interpretation of the results and understand the potential impact in real-world scenarios.

By adhering to these measures, the reliability and validity of the research findings are enhanced, ensuring that the conclusions drawn from this study are well-founded and contribute to the understanding of dependency management and vulnerability scanning in software development.

3.6 Ethical considerations

A software repository was provided by an enterprise, it presented a catalogue of specific software dependencies, each with their unique vulnerabilities. The insights drawn from this information are fundamental to this report and subsequent findings.

There are potential risks associated with the disclosure of these vulnerabilities. Given the sensitive nature of the vulnerabilities and how they can be exploited, disclosing the origin of this information poses a considerable risk. Deliberate decision have been made not to disclose either the identity of the company or the specifics of the repository from which the data was sourced. This is a measure taken in order to safeguard against potential unethical practices and exploitation, ensuring that the knowledge shared in this research is used constructively and responsibly.

4 Implementation

In this project, a custom pipeline was developed that works alongside Dependabot to supplement the handling of vulnerable dependencies in JavaScript and TypeScript projects hosted on GitHub repositories. Dependabot is responsible for scanning the repositories to identify dependencies with known vulnerabilities and, in some cases, updating them with patched versions through pull requests. The custom pipeline utilizes the information provided by Dependabot to further analyze the repository, identifying the files and specific lines where these vulnerable dependencies are used. In cases where Dependabot is unable to create a pull request to resolve a vulnerability, the custom pipeline will generate an issue to ensure that developers are made aware of unresolved security concerns. Both the issue and the enhanced pull request descriptions generated by the pipeline will specify which files and lines the dependencies are used in, enabling developers to address the security risks within their codebase. The source code for the implementation can be accessed from a GitHub repository [34].

4.1 Software Architecture

The software architecture of the custom pipeline is designed to streamline the process of handling vulnerable dependencies in a GitHub repository. This section presents an overview of the architecture, detailing the steps involved in the pipeline to provide a better understanding of its functionality. Following this overview, the software architecture steps are enumerated in the following list:

- Set up Dependabot by creating a .github/dependabot.yml file in the project's root folder.
- Configure the pipeline by creating a .github/workflows/dependency_check.yml file in the project's root folder.
- The pipeline is triggered through new pull requests, pull request merges into a specified branch or push events to that specified branch.
- When a non-merged pull request from Dependabot triggers the pipeline, it runs the .github/scripts/update_pull_request.py script. This script updates the pull request description with the usage information of the dependencies that Dependabot updates.
- If a merged pull request into a specified branch or a push event to that branch triggers the pipeline, it runs the .github/scripts/handle_issue.py script. This script updates an existing issue or creates a new one with usage information of all dependencies found among the Dependabot vulnerability alerts.
- Both .github/scripts/update_pull_request.py and .github/scripts/handle_issue.py uses .github/scripts/scan_files.py to scan all files in the repository and extract usage information.

4.2 GitHub Actions Workflow

The pipeline is executed as a GitHub Actions workflow, defined in the .github/workflows/dependency_check.yml file. Figure 4.2 presents the beginning of the workflow file, defining the pipeline's name as Run pipeline, and specifying its triggers under the on key. The pipeline is triggered by two types of events: pull requests and push events. Pull requests trigger the pipeline if their type is either opened (new pull request) or synchronize (updated pull request). Additionally, the pipeline is triggered by push events to a specified branch, such as the main branch in this example.

```
name: Run pipeline
on:
  pull_request:
    types:
      - opened
      - synchronize
  push:
    branches:
      - main
...
```

Figure 4.2: Workflow triggers.

Figure 4.3 illustrates the pipeline's jobs, which in this case consist of a single job called scan_dependency_usage. As the pipeline runs on all opened and synchronize pull request events, a conditional check ensures that the pull request is created by Dependabot and not merged, or that it is a merged pull request into the specified branch or a push event. This check helps avoid running unnecessary installations and scripts.

```
...
jobs:
  scan_dependency_usage:
    if: (github.event.pull_request.user.login ==
       'dependabot[bot]' && github.event.pull_request.merged !=
       true) || (github.event_name == 'push' ||
       (github.event.pull_request.merged == true &&
       github.event.pull_request.base.ref == 'main'))
...
```

Figure 4.3: Workflow jobs.

When the criteria for running the pipeline are met, the scan_dependency_usage job will execute the steps shown in Fig. 4.4.

- The runs-on: ubuntu-latest action sets up a virtual machine running the latest Ubuntu version on a GitHub server [35].

- The actions/checkout@v2 action checks out the target repository and clones it onto the virtual machine.

- The actions/setup-python@v2 action configures the Python environment with the specified 3.10.11 version [36].

- The PyGithub library (version 1.58.1) is employed in the Python script to interact with the GitHub API [37]. This dependency is installed using pip along with the requirements.txt file.

```
...
jobs:
  scan_dependency_usage:
    if: ...
    runs-on: ubuntu-latest
    steps:
    - name: Check out repository
      uses: actions/checkout@v2

    - name: Set up Python environment
      uses: actions/setup-python@v2
      with:
        python-version: 3.10.11

    - name: Install dependencies
      run: |
        pip install -r .github/scripts/requirements.txt
...
```

Figure 4.4: Workflow steps.

Once the virtual machine and the environment are configured, the next step is to determine if the pipeline has been triggered by an unmerged pull request from Dependabot. If so, the .github/scripts/update_pull_request.py script will be executed, utilizing a GitHub token, repository name, and pull request number as environment variables, as demonstrated in Fig. 4.5.

```
...
 - name: Update pull request
     if: github.event.pull_request.user.login ==
        'dependabot[bot]' && github.event.pull_request.merged !=
        true
     env:
       GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
       REPOSITORY: ${{ github.repository }}
       PULL_REQUEST: ${{ github.event.number }}

     run: |
       python3 .github/scripts/update_pull_request.py
...
```

Figure 4.5: Run script that updates pull request description.

Fig. 4.6 illustrates the subsequent step, where the pipeline checks if it has been triggered by a push event or by a pull request that has been merged into the specified branch (in this example, the main branch). If this condition is met, the .github/scripts/handle_issue.py script will be executed, using a secret token as an environment variable.

```
...
- name: Create/Update issue
  if: github.event_name == 'push' ||
     (github.event.pull_request.merged == true &&
     github.event.pull_request.base.ref == 'main')
  env:
    TOKEN: ${{ secrets.MY_GITHUB_TOKEN }}

  run: |
    python3 .github/scripts/handle_issue.py
```

Figure 4.6: Run script that creates/updates issue description.

4.3 Scan repository for dependencies

In this section, the scan_files.py file will be examined. This Python script is designed to analyze JavaScript and TypeScript files in a given directory for their dependency usage. The primary purpose of this script is to assist developers in understanding how their codebase utilizes imported dependencies. Key aspects of this script, such as scanning and processing import lines, analyzing the usage of dependencies, and generating a report on dependency usage, will be explained.

An aspect of this code is the process of scanning and processing import lines within JavaScript and TypeScript files. This is handled by the process_import_line function, which takes in a line of code, a set of import names, and an import pattern as arguments. The function uses regular expressions to match import statements in the given line and then extracts the imported names or aliases. It ensures that the line is not a comment. If an import statement is found, the function adds the imported names or aliases to the import_name set, which is returned and used in subsequent processing steps. The scan_files function is responsible for iterating through all the files in the given directory and invoking the process_import_line and process_usage_line functions for each line in JavaScript and TypeScript files. It keeps track of import names found in the files and categorizes them based on whether they are part of the node_modules directory or not.

Another aspect is the analysis of the actual usage of dependencies in the codebase. The process_usage_line function is responsible for this task. It takes a line of code, line number, set of import names, a flag indicating if the file is a node_module, and two lists (files_with_dependency and node_modules_with_dependency) as arguments. This function employs regular expressions to seek out any instances of the import names within the code line, excluding import statements themselves. If usage is found, the line number is added to the appropriate list, depending on whether it is a node module or not. The scan_files function also takes care of invoking the process_usage_line function for each line in the JavaScript and TypeScript files, after the import names have been processed.

The final aspect is generating a report on the usage of dependencies within the codebase. This is done by the get_usage_info function. The function takes a list of dependencies, a repository name, a commit SHA, and an optional dictionary of severities as arguments. For each dependency, the function creates an import pattern using regular expressions and then calls the scan_files function to populate the files_with_dependency and node_modules_with_dependency lists. After scanning the files, the function sorts these lists based on the number of lines where the dependency is used. It then generates a detailed report in Markdown format, which includes the following information:

- The dependency name and its severity (if provided).
- The number of files and node modules where the dependency is used.
- The number of files and node modules where the dependency is imported but never used.
- A collapsible list of files and node modules with line numbers where the dependency is used.

This report provides insights into the usage of dependencies in a codebase, which can help developers identify and remove unused dependencies or refactor their code.

4.4 Create and Update Issues

The script described in this subsection is responsible for creating or updating issues related to vulnerable dependencies. It is located in the .github/scripts/handle_issue.py file. Here is a summary of its functionality:

- The script fetches Dependabot alerts using GitHub API.
- It iterates through the alerts, filtering only the open ones, and creates a set of dependencies with their corresponding severities.
- The script generates an issue title and an issue body using the get_usage_info function from the scan_files module.
- It checks for existing open issues with the "Dependabot" label and updates the issue body if the issue already exists or creates a new issue if it does not, in both cases using GitHub API.

4.5 Update pull request description

The .github/scripts/update_pull_request.py script is dedicated to updating pull request descriptions with usage information gathered with the get_usage_info function. Here is the script summarized step by step:

- Initiation of the script involves fetching pull request details via the GitHub API.
- Next, the script pulls out the dependencies related to the pull request from its latest commit.
- Utilizing the get_usage_info function from the scan_files module, a summary report is generated, detailing the usage of each dependency within the pull request.
- Pull request description is updated with the report using GitHub API.

4.6 Summary

In this project, a custom pipeline has been implemented that leverages the power of Dependabot to automatically patch vulnerable dependencies in Node.js and JavaScript projects. By integrating a Python script into a GitHub Actions workflow, the pipeline enhances the pull request descriptions generated by Dependabot, thereby providing developers with more actionable information about the location and usage of vulnerable dependencies in their codebase. This solution helps developers to make informed decisions about updating their dependencies, potentially saving time and effort in manually reviewing and testing the changes. Furthermore, by automating the process of scanning and reporting vulnerable dependencies, the pipeline contributes to improving the overall security of the software being developed.

5 Experimental Setup and Results

This chapter presents the experimental setup and the raw results. These primary results will form the basis for in-depth analysis and discussion in subsequent chapters, aimed at answering the research queries posed in Chapter 1.3.

5.1 Experimental Setup

The evaluation is conducted using a single closed-source project, provided by a large tech firm. This repository was selected based on several factors such as size, complexity, amount of vulnerabilities, and dependency management practices, but an additional significant advantage was direct access to the developers maintaining the repository. This offers a unique opportunity to gain first-hand insights into the implementation's performance and impact.

The implementation is configured to scan through all directories in the repository, including the node_modules directory, which houses all the project's dependencies. As such, it encounters a broad array of coding styles, practices, and patterns. This extensive scanning strategy not only offers a realistic testing environment but also presents a challenge for the implementation, thereby enabling an evaluation of its ability to detect vulnerabilities. The implementation only scans for dependencies that Dependabot has found vulnerable. To enhance the evaluation dataset, the "react" dependency was intentionally included in the scanning list. This inclusion allowed the system to scan not only for dependencies with known vulnerabilities but also for widely used dependencies like "react". This increased the number of imports and usage instances in the codebase, thus providing more data for assessing the implementation's performance.

The following setup procedure is conducted:

1. Create a GitHub repository and add the .github folder from the implementation repository and the repository that should be tested.

2. Enable Dependabot alerts and Dependabot security updates in the GitHub repository [9].

3. Create a GitHub personal access token or regenerate an already existing one and copy the token [38].

4. Create an action secret on the GitHub repository and name it MY_GITHUB_TOKEN and paste the personal access token to the value field [39].

5. To test the react dependency without found vulnerability, .github/scripts/handle_issue.py is updated by temporarily adding "react" to the dependencies list.

6. Push the repository to the main branch in the remote GitHub repository.

After the repository is pushed, the pipeline will automatically run. It generates an issue titled Vulnerable Dependencies, and if able, Dependabot creates pull requests featuring patched dependencies. Both the issue and the pull requests include the dependency usage report. This setup facilitates the answering of research questions outlined in Chapter 1.3.

5.2 Usage Communication

To address the RQ1 research question, the implemented pipeline uses two strategies to communicate dependency usage information. In the first approach, the pipeline identifies vulnerable dependencies through Dependabot alerts, scans the repository for these dependencies, and creates a detailed usage report. This report is then shared with developers in an issue. The second approach is initiated when a new pull request is created by Dependabot. Here, the pipeline retrieves dependencies from the pull request, scans the repository accordingly, and appends the resulting usage information directly to the pull request description.

When the implementation communicates the information via an issue, all dependencies with vulnerabilities are displayed by its name as a header, the vulnerability's severity, and a summary of the vulnerability. If no imports are found, it will be displayed as in Fig. 5.7.

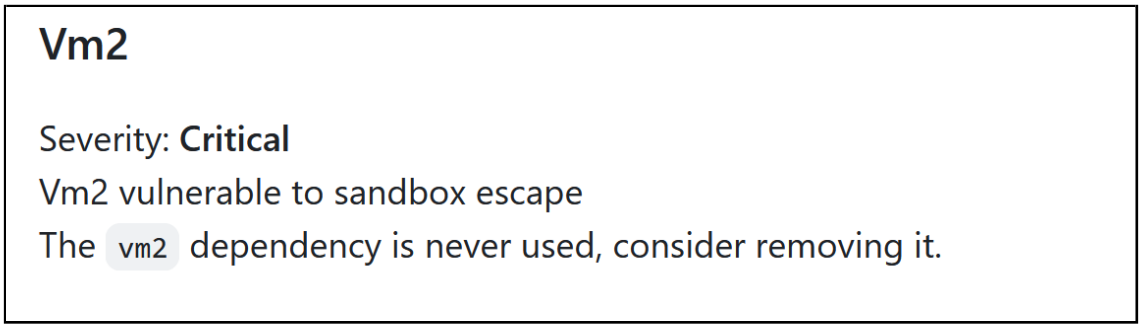

Figure 5.7: Screenshot of how the pipeline displays no import of a dependency.

When the script detects an imported, but unused, dependency, the outcome is displayed as in Fig. 5.8. It creates a collapsible list under Files or Node modules, providing links to the files with the unused import.

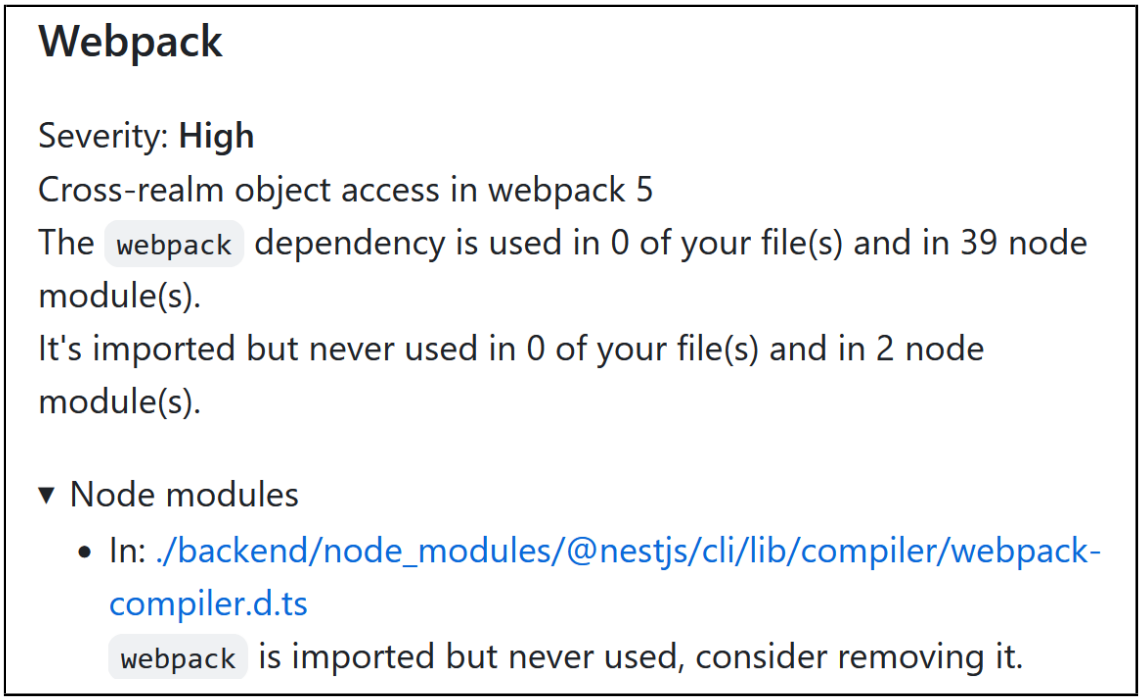

Figure 5.8: Screenshot of how the pipeline displays import but no usage of a dependency.

When the usage of a dependency is discovered, the information is provided in the format as shown in Fig. 5.9. The system presents links to the specific files where the dependency is imported. Furthermore, it highlights the exact lines within these files where the dependency is utilized.

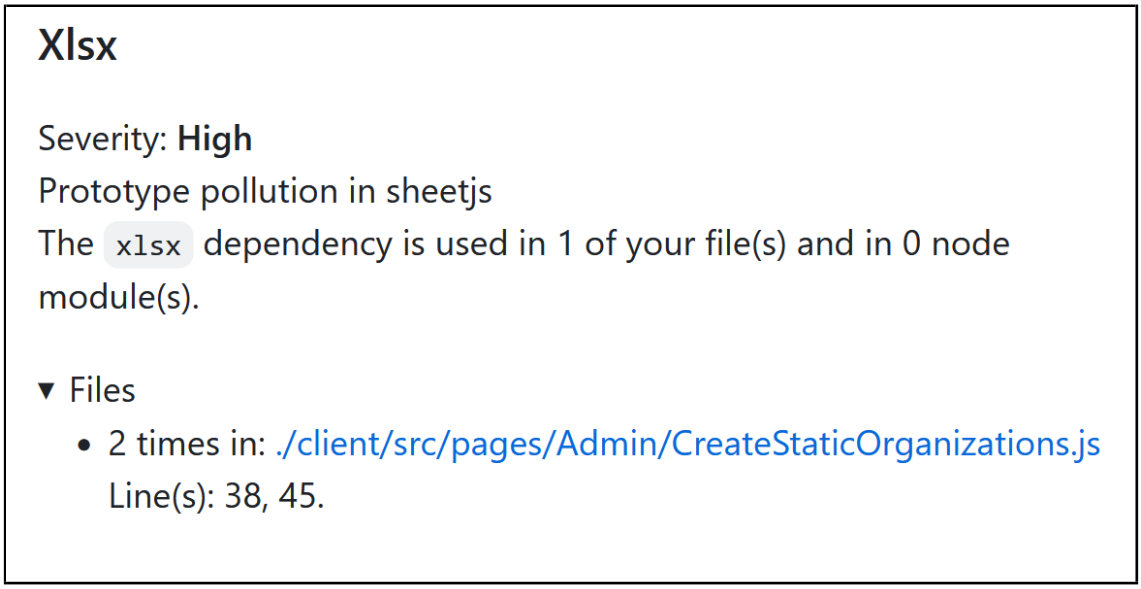

Figure 5.9: Screenshot of how the pipeline displays usage of a dependency.

When Dependabot initiates pull requests with patched updates of dependencies, the script adds Dependabot as a header and puts the original description from Dependabot in its body, as shown in Fig. 5.10.

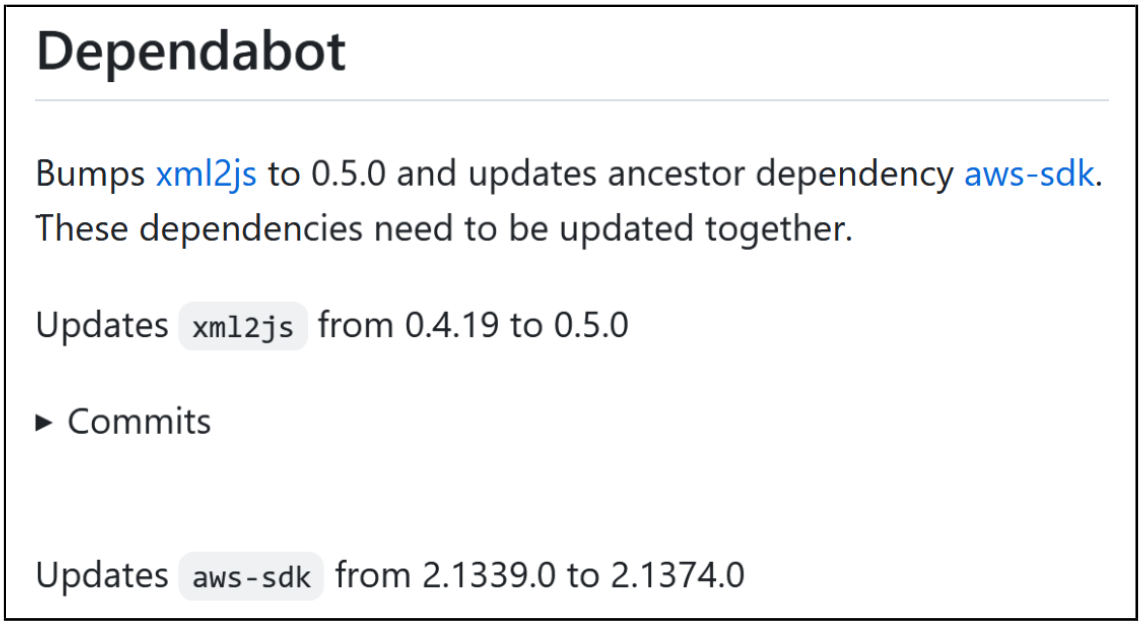

Figure 5.10: Screenshot of pull request from Dependabot.

Building upon the information provided by Dependabot, the script then scans the repository for the dependencies identified in the pull request. It introduces a Usabot header and appends detailed usage information, equivalent to what is included when creating issues, as shown in Fig. 5.11.

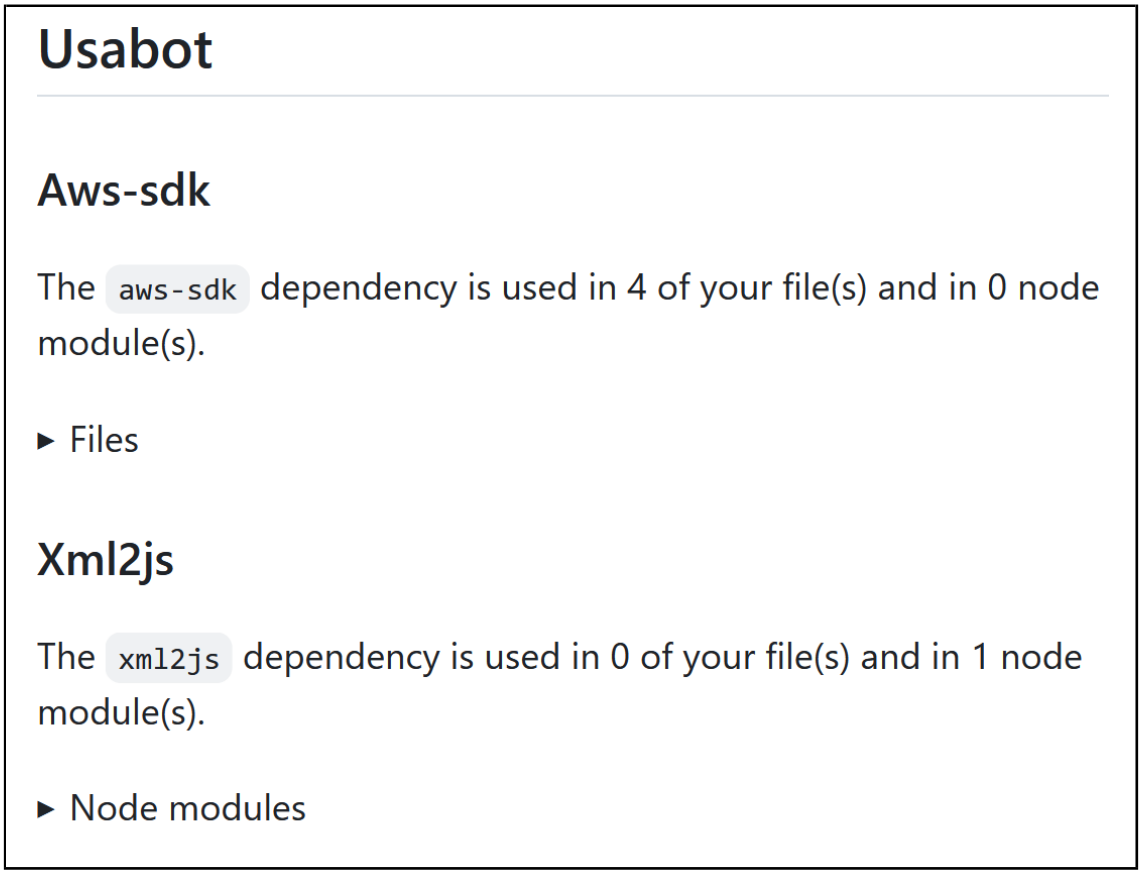

Figure 5.11: Screenshot of pull request enhancement from pipeline.

5.3 Custom Implementation Evaluation

This section is dedicated to addressing RQ2 by evaluating the proficiency of the custom solution in identifying the actual usage of dependencies. It assesses the solution's accuracy in terms of true positives, false positives, and false negatives. The evaluation process consists of several steps, from data collection and performance measurement to a detailed analysis of the findings.

5.3.1 Testing procedure

The testing procedure proceeds as follows:

1. For each dependency listed in the Vulnerable Dependencies issue, manually inspect all files. If the dependencies are truly imported, they are classified as true positives. In cases where they are not imported, they are deemed false positives.

2. Similarly, verify the usage of each dependency in the Vulnerable Dependencies issue. This involves manually checking all files to confirm that dependencies are used in the specified lines. Correct usage results in a true positive, while incorrect usage counts as a false positive.

3. Utilize Visual Studio Code's search across files function [40]. For each dependency in the Vulnerable Dependencies issue, conduct a search using the dependency name. Review all search results to identify any import or usage not covered in the report. Any missed instances count as false negatives.

5.3.2 Results

A total of 115,956 JavaScript and TypeScript files were scanned, and 7 vulnerable dependencies were found by Dependabot. This initial set of vulnerabilities provides a foundation for analyzing the proficiency of the implementation in detecting and addressing vulnerabilities.

Table 5.3 illustrates the number of times dependencies are imported within the codebase. Conversely, Table 5.4 indicates the frequency of each dependency's usage. Both tables also highlight instances where the script incorrectly identifies false negatives or false positives.

Table 5.3: Instances of dependency imports.

Dependency	True Positives	False Positives	False Negatives
class-validator	12	0	0
jsonwebtoken	2	0	0
xlsx	1	0	0
nth-check	2	0	0
webpack	39	0	0
xml2js	3	0	0
vm2	0	0	0
react	110	0	0
Total	**169**	**0**	**0**

Table 5.4: Instances of dependency usage.

Dependency	True Positives	False Positives	False Negatives
class-validator	236	0	0
nth-check	8	0	0
jsonwebtoken	2	0	0
xlsx	2	0	0
webpack	96	26	3
xml2js	3	0	0
vm2	0	0	0
react	275	11	0
Total	**622**	**37**	**3**

6 Analysis

Building on the raw data presented in the previous chapter, this chapter is dedicated to the thorough analysis of these results. This analysis seeks to unravel the performance, strengths, and shortcomings of the implemented pipeline and its communication strategies. The aim is to provide a comprehensive understanding of the pipeline's effectiveness in addressing the research questions laid out in Chapter 1.3.

6.1 Communication Analysis: Dependabot vs. Custom Implementation

This section focuses on how the implemented pipeline communicates information about dependency usage to developers, and more importantly, the added value it brings compared to Dependabot alone.

Dependabot, while effective at identifying vulnerable dependencies and alerting developers through alerts or pull requests, lacks the aspect of providing information about how and where these dependencies are used within the codebase. This presents a challenge for developers, who must manually trace the usage of these dependencies, to gauge the potential impact of a vulnerability. This process can be time-consuming and error-prone, especially in large projects.

The custom implementation addresses this issue by automating the process of tracing the usage of dependencies. It provides detailed information about where a dependency is imported and used within the codebase, directly within the issue or pull request created by Dependabot. This additional information could not only save developers time but also reduce the chances of overlooking vulnerable code. Moreover, by providing this context, the custom implementation helps developers understand the potential impact and risk associated with each vulnerability. This, in turn, aids in informed decision-making regarding the prioritization and remediation of vulnerabilities. It is important to note that not all vulnerabilities carry the same risk. For instance, a vulnerability in a dependency that is widely used across the codebase may pose a greater risk than one in a dependency that is rarely used or not used at all.

In essence, the custom implementation enhances Dependabot by addressing its limitations and adding a layer of context to the vulnerability alerts. By doing so, it improves the effectiveness of vulnerability management and, ultimately, the security of the software.

6.2 Assessing Accuracy in Dependency Usage Identification

An exploration of the custom solution's proficiency and precision in pinpointing actual dependency usage is undertaken in this section. This exploration encompasses an assessment of false positive and false negative instances, signifying inaccuracies in the system's reporting.

The custom solution's primary goal is to correctly identify instances of dependency usage within the codebase. A true positive refers to a successful identification of actual dependency usage, whereas a false positive represents a reported usage that does not truly exist in the codebase. Conversely, a false negative implies that the tool has failed to report an actual usage of a dependency. The calculations, as explained in Chapter 3.4.1, result in the values displayed in Table 6.5. These values indicate the script's effectiveness in identifying import and usage instances, based on the statistical analysis of binary classification. Precision, recall, and F_1-score are employed for this assessment.

Table 6.5: Calculations of recall, precision and F_1-score.

Name	Equation	Imports	Usage
Precision	Eq. 1	1	0.944
Recall	Eq. 2	1	0.995
F_1-score	Eq. 3	1	0.969

In previous chapter, Table 5.3 displays the number of dependency imports identified by the implementation. As shown in the table, the implementation achieved perfect performance for dependency import detection, with 169 true positives and no false positives or false negatives. This indicates that the implementation accurately identified all instances of dependency imports without any mistakes, resulting in a precision, recall, and F_1-score of 1. In Table 5.4 the performance, of the implementation in detecting instances of dependency usage, is illustrated. It identified 659 instances of dependency usage, out of which 37 were false positives, and it missed 3 usages. Despite these inaccuracies, the implementation demonstrated satisfactory performance in identifying dependency usage.

The precision score signifies that the implementation correctly identified 94.4% of all detected instances as actual dependency usage. The high recall score indicates that the implementation successfully detected 99.5% of all actual instances of dependency usage. The F_1-score, which is the harmonic mean of precision and recall, shows that the implementation maintained a balance between precision and recall, providing an overall strong performance in identifying dependency usage.

6.2.1 False Statements

In this project's context, false statements refer to erroneous conclusions drawn by the implementation regarding the presence of a dependency in the codebase. These misinterpretations can take two forms: false positives and false negatives.

False positives occur when the implementation incorrectly identifies non-usage of a dependency. This can happen in two instances: during the import of a dependency or during the supposed usage of the dependency. By analysing all false positives, two different reoccurring patterns were noticed. The first pattern, as illustrated in Fig. 6.12, arises when the import name, webpack, is correctly identified (line 1), and is discovered within a string (line 3), where the import name is encompassed by word breaks. It is important to note that correct identification would have been made in cases where similar strings such as require('webpack-plugin') or require('webpacking') are encountered.

```
const webpack = require('webpack');
const resolve = require('resolve');
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

Figure 6.12: Example of a false positive in dependency usage.

The second pattern, displayed in Fig. 6.13, occurs when an import name, which would have been correctly recognized under normal circumstances, is instead enclosed within a multi-line comment. Note that the current implementation correctly identifies instances within single-line comments (prefixed by //) and multi-line comments that begin each line with an asterisk. However, it fails to recognize the non-usage within the standard multi-line comments without asterisks at the start of each line, thereby resulting in false positives.

```
import { useEffect, useState } from 'react'
...
/*
const [approvedMembers, setApprovedMembers] = useState([])
*/
...
```

Figure 6.13: Example of a false positive in dependency usage.

False negatives refer to instances where the implementation fails to identify the actual usage of a dependency. In the experiment, only three instances of false negatives were found. All three instances of false negatives followed the same pattern, illustrated in Fig. 6.14. Here, the webpack dependency is imported in an unconventional way that combines the use of import and require. This unusual import syntax results in an incorrect identification of the entire expression webpack = require as the import name, leading to the inability to match subsequent usages of webpack in the code.

```
import webpack = require('webpack');
...
...webpack.Configuration...
```

Figure 6.14: Example of a false negative in dependency usage.

7 Discussion

Reflecting on the progress and key findings of our study, we delve into a comprehensive analysis of the work conducted. This project aimed to improve the understanding of dependency usage in JavaScript and TypeScript codebases by developing a custom implementation that not only detects dependencies but also tracks their usage. The discussion will revolve around the outcomes and limitations of the system developed, how it compared with other existing tools, the justification for choosing Dependabot as the central tool, and the challenges faced in analyzing open-source projects. We strive to present an objective evaluation of our work and how it contributes to the existing body of knowledge on dependency management in software development.

7.1 Limitations

The custom implementation enhances Dependabot's functionality by providing detailed information about the usage of vulnerable dependencies, yet it is important to acknowledge inherent limitations. The implementation has been primarily tested and evaluated on a relative small set of different of dependencies, therefore its proficiency may vary when dealing with different usage patterns or more complex dependencies. Although the analysis of the node_modules directory offers a broader testing environment, an element of uncertainty persists.

Additionally, the study leans on manual analysis to identify dependency usage, which could overlook certain instances due to human error or oversight, potentially impacting the accuracy of the ground truth data against which we measure the implementation's performance. This risk could be mitigated by involving multiple evaluators and resolving discrepancies through consensus. Our study does not consider the real-time performance of the implementation, leaving factors such as speed and resource consumption during the scanning process unaccounted for. These factors could influence the practical application of our solution.

The efficacy of our implementation can also be inadvertently influenced by the performance of Dependabot, especially if Dependabot encounters issues like inaccurate vulnerability alerts or dependency updates. As a result, our results should be interpreted considering this potential source of error. Moreover, our custom implementation is constrained by a character limit for issue creation, which could present difficulties when dealing with large repositories containing numerous dependencies, leading to truncated or incomplete issue descriptions.

Finally, while our custom implementation provides additional context for each vulnerability, developers still bear the responsibility of assessing potential impact and risk. This underscores the continued importance of informed decision-making and manual analysis, reminding us that our implementation aids in the process, but does not eliminate these requirements.

7.2 Segregated Detection of Imports and Other Usage

This report involves the detection and analysis of both imports and usage of dependencies in JavaScript and TypeScript codebases. One may question why the results of import detection and all other usage are separated in the reporting, there are reasons for this segregation.

The custom pipeline's strategy, is to use the "import name" from import statements to detect other usages in the code. The reliance on the import name as a key identifier links to the success of detecting other usages. This strategy might introduce a specific form of false negatives into our results, specifically in cases where the import detection fails or inaccurately identifies the import name. Any usage of such a dependency could be overlooked, as the detection process would be searching for an incorrect import name. Such usage would not be identified and would be classified as false negatives. However, these false negatives represent instances where the dependency usage probably would have been detected, provided the initial import name was correctly identified.

7.3 Detecting Usage

A limitation of the custom implementation was identified, in detecting dependencies usage within JavaScript and TypeScript code. The system misses certain usage instances, which is vital to consider when interpreting the results. In Fig. 7.15, the express dependency is imported and used to instantiate the app object. Subsequently, app is used to call the use method. The present implementation may fail to recognize these as instances of express usage since they do not directly involve the express identifier.

```
import express from 'express'
import logger from 'morgan'

const app = express()
const PORT = 3000

app.use(logger('dev'))
app.use(express.static('public'))
```

Figure 7.15: Indirect dependency usage missed by the current implementation.

Such indirect usages are prevalent in JavaScript and TypeScript, due in large part to the extensive use of Object-Oriented Programming (OOP) paradigms. Our current solution might not detect these scenarios, potentially leading to an underestimation of a dependency's usage.

This inherent limitation in the current implementation is an intentional feature. The complexity of tracking a dependency across variable assignments and method calls in each file would significantly increase computational requirements. The process involves not only identifying the original import statement but also tracking each instance where the imported module is assigned to a new variable or passed as a parameter. This introduces additional complexity due to the dynamic and flexible nature of JavaScript and TypeScript. Implementing this level of analysis would result in an exponential increase in computing resources and processing time, making it impractical for large code repositories. Consequently, the pipeline runtime would be substantially longer, hindering the scalability of the solution. Thus, our implementation makes a trade-off between accuracy and efficiency, by focusing on the most direct and common usage patterns to maintain acceptable performance levels while still delivering useful insights on dependency usage.

7.4 Emphasizing Recall Over Precision

In our approach to developing a custom implementation supplementing Dependabot, an essential element of our design strategy is to prioritize the detection of false positives over false negatives. In the language of evaluation metrics, this priority translates to favoring recall, the ability to find all relevant instances, over precision, the proportion of true positive against all positive results.

False positives, in this context, would correspond to instances where the tool detects dependency usage in the codebase when, in fact, there is none. On the other hand, false negatives denote situations where the tool fails to recognize the actual usage of a dependency. The latter case can lead to significant problems since it suggests that a dependency is not used, thus encouraging developers to remove it. If done, it could result in bugs in the code if the dependency was, in fact, necessary. It is preferable to deal with false positives because they are much less likely to disrupt the codebase's stability. In these instances, the tool would recommend further inspection of a dependency that might not be necessary. Although this might cause minor inconvenience to developers due to the additional effort required to validate the usage, it will not impact the software's operational integrity. Moreover, developers can utilize these instances to review and ensure that their code is free from extraneous dependencies, thereby improving code quality.

In contrast, overlooking an important dependency, which is a false negative case, could directly lead to a critical failure in the system. This would not only be more difficult to debug but could also compromise the security of the system if the overlooked dependency contains a vulnerability. Therefore, while it is important to reduce both types of errors, it is crucial to focus on minimizing false negatives to maintain the robustness and security of the system.

7.5 Difficulties in Manually Confirming False Statements

The task of manually verifying the performance of the implementation, notably in terms of distinguishing false positives and negatives, introduces certain complexities. The relative ease of spotting these discrepancies can vary, potentially skewing the perception of the results.

False positives, characterized as instances where the implementation incorrectly recognizes a non-usage of a dependency as usage, are typically more straightforward to detect in manual analysis. These instances can be ascertained by contrasting the alleged usage with the codebase, if no such usage is present it can be promptly classified as a false positive. However, the potential exists for such occurrences to distract developers and misdirect resources away from actual vulnerabilities, underscoring the need for accurate detection methods.

On the other hand, false negatives, which arise when the implementation overlooks an actual dependency usage, are more challenging to pinpoint. They represent scenarios of missing information, and spotting the absence of something can be complicated. This necessitates an intimate knowledge of the codebase and exceptional attention to detail. The inherent difficulty could result in a lower reporting of false negatives, potentially presenting a distorted view of the implementation's performance. This could lead to undetected vulnerabilities, jeopardizing the security of the system.

To counter these complexities, the implementation was designed with an intent to capture all instances of dependency usage, under the premise that it is preferable to encounter some false positives rather than miss true positives, leading to false negatives. The design decision was based on the recognition that, in the context of security, an overlooked vulnerability (a false negative) can have far more serious implications than an incorrect alert (a false positive).

Given the challenges presented, a high level of meticulousness and caution was essential during the manual analysis. The reliability of the results could in the future be amplified by involving multiple evaluators to cross-validate the observations, offering thorough training in code analysis, and considering the use of automated tools to aid in the detection of false negatives. By addressing these concerns, a more accurate evaluation of the implementation's performance can be achieved. This proactive and multifaceted approach would foster an environment of continuous improvement and a relentless pursuit of precision in vulnerability detection and dependency management.

7.6 Exploration of Alternative Approach: Esprima Library

During the development of this project, an alternative approach for analyzing the usage of dependencies was also explored, leveraging a JavaScript and TypeScript parsing library Esprima [41]. The rationale behind considering Esprima was its robustness and capacity to perform detailed syntactic analysis of JavaScript code, potentially leading to more accurate results compared to a regular expression-based approach.

Esprima, being an ECMAScript parsing infrastructure for multipurpose analysis, is widely known for generating an Abstract Syntax Tree (AST) from JavaScript source code. By providing a hierarchical, tree-like structure, it facilitates more precise manipulation and interpretation of code. This characteristic made it an enticing prospect for our dependency usage analysis task.

However, upon implementation and subsequent testing, it was found that Esprima failed to parse approximately 14.7% of the files in the repository. This failure rate introduced a considerable degree of inaccuracy, as a significant portion of the repository could not be examined for dependency usage. In essence, the data from these files were excluded from the final report, leading to an incomplete and potentially misleading assessment.

After weighing the advantages and disadvantages, the decision was made to proceed with the regular expression-based approach. Despite its simplicity, regular expressions provided a more comprehensive and reliable analysis for this specific project. They successfully parsed all the files in the repository, ensuring a complete evaluation of dependency usage.

Although the regular expression method might seem less advanced compared to a technique like Esprima which uses an Abstract Syntax Tree (AST), its ability to perform a comprehensive analysis with fewer instances of parsing errors made it invaluable for our purposes. It underscores a key learning from this project: the choice of tools and methods significantly depends on the specific requirements and constraints of a task. Accuracy is not solely about depth of analysis, but also about the completeness and reliability of the data being analyzed. In this case, regular expressions outperformed a more specialized tool in terms of delivering accurate, comprehensive results.

7.7 Choice of Dependabot over Other Tools

During the literature review, a few other tools that offer dependency management capabilities were explored, including Snyk, Renovate, and Retire.js [42, 43, 44]. Their key features were compared with Dependabot to explain why Dependabot was chosen as the tool of choice for this project.

Dependabot, a feature provided by GitHub, is a free and open-source tool that helps developers manage dependencies within their projects. It is designed to keep projects secure by scanning for outdated or insecure dependencies and automatically creating pull requests to update them. One of the major advantages of Dependabot is its integration with GitHub. This integration simplifies the process of setting up workflows and allows for easier integration of the developed implementation. Dependabot's popularity and extensive community support are other reasons for this choice. The tool has been thoroughly tested and proven to be reliable and effective in various scenarios. This extensive user base also serves as a vast testing ground for potential issues, ensuring that any discovered problems are quickly addressed. Dependabot is actively maintained by GitHub, ensuring it receives regular updates to handle new dependency management scenarios and that it is promptly patched when any issues are discovered.

While Snyk, Renovate, and Retire.js each offer their unique strengths in dependency management, the deciding factors were the ease of script implementation with Dependabot and GitHub, and its status as the most widely used bot. is is important to note that the choice of tool greatly depends on the specific needs and constraints of a project. The factors that led to choosing Dependabot for this project may not necessarily apply to all software development projects [12].

7.8 Challenges in Open-source Projects

Many larger open-source repositories employ Dependabot or other automated tools for dependency management. These bots handle the updating and patching of dependencies, which could interfere with the experiment conducted in this study. The previous actions of these bots could affect the current state of dependencies in the repository, making it difficult to isolate the impact and effectiveness of the custom pipeline implemented in this project.

In some cases, open-source projects may have removed Dependabot for reasons that are not documented or readily apparent. This introduces an element of uncertainty into the analysis. The absence of Dependabot could impact how dependencies are managed and updated in the repository, potentially complicating the assessment of the custom pipeline's performance.

The nature of open-source projects introduces additional unknown variables into the study. Factors such as the number of contributors, the frequency of commits, the scale of the project, and the project's overall management practices can vary widely among different open-source projects. These factors could influence how dependencies are used and managed in the project, adding complexity to the analysis and interpretation of the results.

Given these challenges, it is important to approach the analysis of open-source projects with caution and flexibility. While these factors add complexity, they also provide an opportunity to gain insights into the real-world application and challenges of managing dependencies in diverse and dynamic project environments.

7.9 Time Constraints

Given the time constraints of this study, a comprehensive application of the custom pipeline to multiple open-source projects was not feasible. This limitation was primarily due to the complexity of manually analyse open-source projects, as well as the challenges outlined above.

However, the application of this custom pipeline to open-source projects presents a valuable opportunity for future research. With more time and resources, the pipeline could be applied to a broader range of open-source repositories. This would allow for a more robust evaluation of its performance, offering insights into how it handles different project structures, sizes, and dependency management practices. It could also provide valuable real-world feedback from the open-source community, potentially highlighting areas for further improvement and refinement of the implementation.

A noteworthy aspect to consider involves the improvement of the custom pipeline implementation by successfully addressing a couple of identified errors. During the controlled experiment, we encountered issues related to multi-line comments and dependencies found within strings. These issues resulted in a somewhat less robust implementation than we would ideally want.

These errors, however, are far from insurmountable. With additional time beyond the constraints of this study, they can be effectively resolved with relatively simple modifications to the source code. Despite recognizing these problems during the experiment, the timeline of this project did not permit adequate space for both rectifying the issues and re-performing the experiment to evaluate the improved implementation.

8 Conclusion

This project developed and evaluated a pipeline to extend Dependabot's capabilities in managing dependencies and their vulnerabilities in JavaScript and TypeScript projects. The custom implementation identifies the import and usage of dependencies within a codebase, providing an enhanced understanding of how each dependency is used and the potential risks associated with them.

The results of the study bear relevance to both the industry and academia. From an industry perspective, the implementation improves the efficiency of vulnerability management in software projects, potentially saving significant development time and enhancing the overall security of the projects. In terms of academia, the research provides insights into the challenges associated with automated dependency management and the effectiveness of existing tools, contributing to the broader field of study on software maintenance and security.

The findings, while particularly applicable to JavaScript and TypeScript projects, hint at the broader potential of this approach. With further development and adaptation, the implementation could be extended to handle other programming languages and package managers, enhancing the general applicability of the results.

However, there are aspects of the project that could have been done differently for possibly improved outcomes. For instance, a broader and more diverse set of repositories could have been used in the evaluation to further challenge the implementation and provide a more comprehensive view of its performance. Furthermore, the evaluation of the system's real-time performance, including factors such as speed and resource consumption, could have provided additional insights into its practical applicability.

8.1 Future work

If this project were to be continued, there are several directions that could be pursued. Firstly, the adaptation of the implementation to handle other programming languages and package managers would be a crucial step. This could involve extending the current pipeline or developing new pipelines using similar methodologies.

Moreover, enhancing the precision of dependency identification could be another focus. This might involve refining the algorithm or exploring machine learning-based approaches to improve accuracy. We could also look into why some open-source projects decide not to use Dependabot. It would be interesting to see if a customized version of our system could meet their requirements. This could be a worthwhile area to explore.

Lastly, comprehensive performance evaluation, including aspects such as speed, resource consumption, and scalability, could be carried out to further understand the practical implications of the implementation. This could also include a more thorough investigation into the impact of false positives and false negatives and strategies to minimize them.

References

[1] N. Harrand, *Software Diversity for Third-Party Dependencies*, 2022. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1650630&dswid=5473

[2] "Log4shell: Redefining the web attack surface - NDSS symposium." [Online]. Available: https://www.ndss-symposium.org/ndss-paper/auto-draft-277/

[3] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 1–36, feb 2018. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9589-y

[4] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. New York, New York, USA: ACM Press, may 2018, pp. 181–191. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3196398.3196401

[5] H. Berghel, "Equifax and the latest round of identity theft roulette," *Computer*, vol. 50, no. 12, pp. 72–76, dec 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8220474/

[6] I. Journal, "IRJET- analysis of open source node.js vulnerability scanners," *IRJET*, jan 2020. [Online]. Available: https://www.academia.edu/44237495/{IRJET_Analysis_of_Open_Source_Node_js_Vulnerability_Scanners}?sm=b

[7] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2016, pp. 470–481. [Online]. Available: http://ieeexplore.ieee.org/document/7476667/

[8] "[PDF] automating dependency updates in practice: An exploratory study on GitHub dependabot |semantic scholar." [Online]. Available: https://www.semanticscholar.org/reader/1205202c99d80547df80c19eb0f9781471f40f96

[9] "Keeping your supply chain secure with dependabot - GitHub docs." [Online]. Available: https://docs.github.com/en/code-security/dependabot

[10] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 1–34, may 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9521-5

[11] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "[1808.09753] vulnerable open source dependencies: Counting those that matter," *arXiv*, aug 2018. [Online]. Available: https://arxiv.org/abs/1808.09753

[12] M. Wyrich, R. Ghit, T. Haller, and C. Muller, "Bots don't mind waiting, do they? comparing the interaction with automatically and manually created pull requests," in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*. IEEE, jun 2021, pp. 6–10. [Online]. Available: https://ieeexplore.ieee.org/document/9474402/

[13] P. Johnson, R. Lagerstrom, M. Ekstedt, and U. Franke, "Can the common vulnerability scoring system be trusted? a bayesian analysis," *IEEE transactions on dependable and secure computing*, vol. 15, no. 6, pp. 1002–1015, nov 2018. [Online]. Available: https://ieeexplore.ieee.org/document/7797152/

[14] Nist. Vulnerability metrics. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[15] "NVD - CVSS v3 calculator." [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

[16] "Downloads |CVE." [Online]. Available: https://www.cve.org/Downloads

[17] "History |CVE." [Online]. Available: https://www.cve.org/About/History

[18] "NVD - CVEs and the NVD process." [Online]. Available: https://nvd.nist.gov/general/cve-process

[19] "CWE - about - CWE history." [Online]. Available: https://cwe.mitre.org/about/history.html

[20] "NVD - CVE-2021-44228." [Online]. Available: https://nvd.nist.gov/vuln/detail/{CVE}-2021-44228

[21] "CWE - CWE-917: Improper neutralization of special elements used in an expression language statement ('expression language injection') (4.11)." [Online]. Available: https://cwe.mitre.org/data/definitions/917.html

[22] "Getting started with GitHub documentation - GitHub docs." [Online]. Available: https://docs.github.com/en/get-started

[23] "GitHub actions documentation - GitHub docs." [Online]. Available: https://docs.github.com/en/actions

[24] "GitHub REST API documentation - GitHub docs." [Online]. Available: https://docs.github.com/en/rest?{apiVersion}=2022-11-28

[25] "Dependabot." [Online]. Available: https://github.com/dependabot

[26] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2021, pp. 254–265. [Online]. Available: https://ieeexplore.ieee.org/document/9463148/

[27] About dependabot alerts. [Online]. Available: https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts

[28] “Viewing and updating dependabot alerts - GitHub docs.” [Online]. Available: https://docs.github.com/en/code-security/dependabot/dependabot-alerts/viewing-and-updating-dependabot-alerts

[29] “Configuration options for the dependabot.yml file - GitHub docs.” [Online]. Available: https://docs.github.com/en/code-security/dependabot/dependabot-version-updates/configuration-options-for-the-dependabot.yml-file

[30] “GitHub advisory database.” [Online]. Available: https://github.com/advisories

[31] G. Inc. Browsing security advisories in the github advisory database. [Online]. Available: https://docs.github.com/en/code-security/security-advisories/global-security-advisories/browsing-security-advisories-in-the-github-advisory-database

[32] “Editing security advisories in the GitHub advisory database - GitHub docs.” [Online]. Available: https://docs.github.com/en/code-security/security-advisories/global-security-advisories/editing-security-advisories-in-the-github-advisory-database

[33] “FAQ |elicit.” [Online]. Available: https://elicit.org/faq

[34] “isakcarlsson/usabot.” [Online]. Available: https://github.com/isakcarlsson/Usabot

[35] “Ubuntu server documentation |ubuntu.” [Online]. Available: https://ubuntu.com/server/docs

[36] “3.10.11 documentation.” [Online]. Available: https://docs.python.org/3.10/

[37] “PyGithub —PyGithub 1.58.1.dev12+g300c5015 documentation.” [Online]. Available: https://pygithub.readthedocs.io/en/latest/

[38] “Creating a personal access token - GitHub enterprise server 3.4 docs.” [Online]. Available: https://docs.github.com/en/enterprise-server@3.4/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token

[39] “Encrypted secrets - GitHub docs.” [Online]. Available: https://docs.github.com/en/actions/security-guides/encrypted-secrets

[40] “Basic editing in visual studio code.” [Online]. Available: https://code.visualstudio.com/docs/editor/codebasics#_search-across-files

[41] “Esprima —esprima master documentation.” [Online]. Available: https://docs.esprima.org/en/latest/

[42] “User docs - snyk user docs.” [Online]. Available: https://docs.snyk.io/

[43] “Renovate docs |renovate docs.” [Online]. Available: https://docs.renovatebot.com/

[44] “Retire.js.” [Online]. Available: https://retirejs.github.io/retire.js/