The Journal of Systems & Software 203 (2023) 111752

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Empirical analysis of security-related code reviews in npm packages ☆

Mahmoud Alfadel [a,*], Nicholas Alexandre Nagy [b], Diego Elias Costa [c], Rabe Abdalkareem [d], Emad Shihab [b]

[a] University of Waterloo, Canada
[b] Concordia University, Canada
[c] University of Quebec in Montreal, Canada
[d] Omar Al-Mukhtar University, Libya

ARTICLE INFO

Article history:
Received 2 July 2022
Received in revised form 3 May 2023
Accepted 15 May 2023
Available online 19 May 2023

Dataset link: https://zenodo.org/record/7538187#.Y9IYE-zMKEs

Keywords:
Open source software
Third-party package
Code review
Security

ABSTRACT

Security issues are a major concern in software packages and their impact can be detrimental if exploited. Modern code review is a widely-used practice that project maintainers adopt to improve the quality of contributed code. Prior work has shown that code review has an important role in improving software quality, however, in-depth analyses on code review in relation to security issues are limited.

Therefore, in this paper, we aim to explore the role of code review in finding and mitigating security issues. In particular, we investigate active and popular npm packages to understand what types of security issues are raised during code review, and what kind of mitigation strategies are employed by package maintainers to address them. With pull requests (PRs) being the medium of code review under study, we analyze 171 PRs with raised security issues. We find that such issues are discussed at length by package maintainers. Moreover, we find that code review is effective at identifying certain types of security concerns, e.g., *Race Condition*, *Access Control*, and *ReDOS*, as dealing with such issues requires in-depth knowledge of the project domain and implementation specifics. Interestingly, we also observe that some projects have automated tools integrated in the project development cycle, which enhances the identification of frequent cases of certain security issues. When analyzing how maintainers respond to the raised security issues, we find that most of the issues (55%) are frequently addressed and mitigated. In other cases, security concerns ended up not being fixed or are ignored by project maintainers. Leveraging our findings, we offer several implications for project maintainers to support the role of reviewing code in finding and fixing security concerns.

1. Introduction

Security vulnerabilities have a large negative impact when found in software packages. In fact, the impact of these vulnerabilities is magnified if they are identified in production, i.e., after the package's version is released, as the new version of the package will be used in a wide number of other software projects, which increases the chance for exploitation. An example of that is the Equifax cybersecurity incident, caused by a web-server vulnerability in the Apache Struts library, which led to the illegal access of sensitive information from almost half of the US population (143 million American citizens) (Equifax, 2021). Such examples show that package maintainers must be very rigorous when checking their code for security issues and vulnerabilities, in order to reduce their impact as much as possible on the software ecosystem that depends on them.

Modern code review is a well-adopted practice in industrial and open source projects, especially with the support of the pull-based development model, for the purpose of ensuring software development quality (McIntosh et al., 2016; Alfadel et al., 2021b). Previous research (e.g., McIntosh et al. (2016) and Dey and Mockus (2020)) provided evidence of the effect of the code review process on the overall software quality level. For example, Dey and Mockus (2020) examined the impact of code review characteristics on accepting pull requests (PRs) in the npm packages, and found that some measures (e.g., PR author and reviewer experience) could influence the PR quality. A recent study by Paul et al. (2021) built a regression model on the Chromium project to identify factors that are associated with successfully identifying vulnerabilities from reviews. They found, for example, that the number of directories under review correlates negatively with identifying vulnerabilities. Given that understanding the effect of code review specifically in relation to software security issues

☆ Editor: Kelly Blincoe.

* Corresponding author.
E-mail addresses: malfadel@uwaterloo.ca (M. Alfadel), Nicholas.a.nagy@protonmail.com (N.A. Nagy), diego.costa@concordia.ca (D.E. Costa), Rabe.abdalkareem@carleton.ca (R. Abdalkareem), eshihab@encs.concordia.ca (E. Shihab).

https://doi.org/10.1016/j.jss.2023.111752
0164-1212/

is critical, our study aims to increase the awareness of project maintainers and researchers to the role of code review in identifying and dealing with security issues in open source projects. In addition, studying this can help us better understand how security issues are being tackled during code review by project maintainers.

Therefore, to shed light on the role of code review from a security perspective, our study aims to analyze code reviews in a set of 10 active, mature, and popular JavaScript GitHub projects from the npm ecosystem. For those projects, we mine the security issues being discovered within PRs of the projects' history, and manually examine more than 4000 review comments in the projects.

We set out to study three Research Questions (RQs). In the first stage of our study, we analyze how often security issues are identified during code review (RQ_1). Out of the 10 studied projects, we find 9 projects in which security issues are raised during code review, with 171 PRs containing evidence of security discussion. Moreover, we observe that such issues are raised in a small fraction of PRs (0.11%–3.56%), affecting also a small fraction of project files (0.25%–3.63%). However, project maintainers discuss the issues at length in the PRs, i.e., 4.82%–28% of all PR comments are related specifically to the security related concerns.

Also, we conduct in-depth manual analysis to understand the types of security issues that package maintainers discover and discuss during code review (RQ_2). Our manual investigation shows that the identified security issues in the projects belong to 14 types. Specific types of these issues are more common across the projects, e.g., Race Condition, Access Control, Sensitive Data Exposure, XSS, Documentation, and Overflow.

Finally, we analyze how package maintainers handle and respond to the identified security issues (RQ_3). We find that the majority (54.96%) of the raised security issues are fixed and mitigated. However, many of the issues seem to be considered as having low threat to the projects (28.65%). In a few cases, we find that the project maintainers decided not to fix the issues (8.18%) or even respond to them (4.67%).

Novelty of the study. Prior research has also touched on the topic of determining the types of vulnerabilities that are able to be identified during code review in open source applications (e.g., Bosu et al. (2014) and Bosu and Carver (2013)). However, we add to the previous work by looking at how project maintainers handle and respond to the identified security issues. Furthermore, we compare issues identified during code review to post-release security vulnerabilities (advisories) that have been reported after the package release production. Such comparison will help us understand whether there are certain types of issues that code review can be effectively employed to identify. Based on our investigation, we offer some suggestions and implications that support the role of code review for the security of open-source packages.

Paper organization. The rest of the paper is organized as follows. Section 2 describes our study methodology. Section 3 presents the results of our study. Section 4 presents our discussion and how our findings lead to implications for practitioners and future research. Section 5 presents the related work. Section 6 presents the threats to validity. Section 7 concludes our paper.

2. Methodology

The main goal of our study is to examine the role that code review plays in identifying and fixing security issues that exist in open source software projects. To achieve our goal, we resort to analyzing the discussion of code reviews throughout the lifetime of open source projects. Our study focuses on analyzing projects that have been published as packages in the Node Package Manager (npm ecosystem). We mine and analyze Pull-Requests (PRs) that exist in the projects. Developers use the PR feature in their GitHub projects as a platform for code reviews. To this end, we first (A) collect a representative set of GitHub projects. Then, (B) we identify candidates for PRs with security-related reviews. Finally, we (C) manually validate the identified PRs, which we use for our study analysis. Fig. 1 provides an overview of our general approach, detailed in the remainder of the section.

2.1. Project selection

We analyze code reviews in JavaScript projects that develop packages published in the npm package ecosystem. We chose to focus on JavaScript due to its wide popularity amongst the development community as JavaScript has consolidated itself as the most popular programming language (Stack Overflow, 2021; Alfadel et al., 2020). The npm ecosystem is the largest software package ecosystem to date, surpassing 1.9M packages published in the ecosystem (npm - Libraries, 2020; Abdalkareem, 2017), with popular packages being used in thousands of program applications (Zerouali et al., 2021). Its popularity and reach, makes the npm ecosystem a prime target for attackers, and maintainers of JavaScript packages have to act fast to identify and remediate software vulnerabilities before they are exploited (Alfadel et al., 2021a; Zimmermann et al., 2019). Consequently, npm has a well-renowned advisory database for reporting security vulnerabilities that affect its projects (npm, 2020).

We collect all npm projects available in the npm advisories dataset (npm, 2020). These projects have some level of popularity and are known to have concerns about security as vulnerabilities have been identified and remediated in their code base.

Our initial dataset contains 1219 unique projects. To begin filtering these projects to a more feasible dataset to work with, we wrote a script to aid in this process. Then, using this automated script, we filter out projects that do not have links to their repositories, as identified with the npm registry, since we have no way of tracking the history of their code review. Of these projects, only 666 projects had links to GitHub repositories. To further curate the dataset for our manual analysis, we apply a filtration process on the projects. Using our automated scripts, we select projects that satisfy all the following filtration criteria:

- *Security concern*: we choose projects that have a minimum number of 2 vulnerability advisories, as we want to include projects that have had a history of identifying vulnerabilities and hence, should be discussing security in their development history (Walden, 2020). Applying this filtration step left us with a dataset of 89 projects.
- *Popularity*: we choose projects with over 10,000 downloads per month, as popular projects are critical to the community, and many developers rely on them for their software development projects (Software, 2019). Upon applying this criteria, we were left with 67 projects.
- *Recent activity*: we choose projects with at least ten commits made in the last month prior to the time of collecting our dataset (i.e., August, 2021), as we want to avoid projects that are no longer active or relevant. Relevancy and activity are essential ways to ensure that the projects we are using for our analysis are current and modern (Kalliamvakou et al., 2014). This step ends us with 37 projects.

It is important to note that our study analysis is very time-consuming, given that such projects contain thousands of PRs, and each PR requires a significant time investment to recognize and understand the context of the security issues being identified. As a result, to make the study feasible, we draw the line at the top 10 most active projects, as this is inline with prior studies that

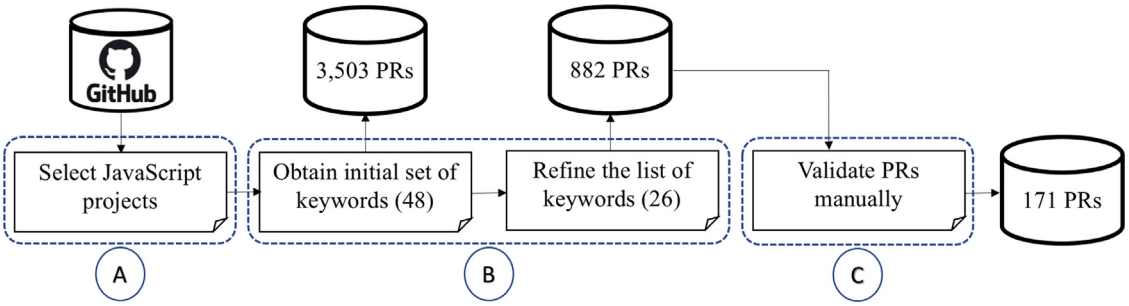

Fig. 1. An overview of our study approach.

Table 1
Overview of projects.

Project	Project description	Language	Age (in years)	# PRs
Marked	A low-level compiler for parsing markdown without caching or blocking for long periods of time (marked - npm, 2021).	JavaScript, HTML	10 years	758
Moment	A lightweight date library for parsing, validating, manipulating, and formatting dates (moment - npm, 2021).	JavaScript	10 years	1,812
Parse-server	An open source backend that can be deployed to any infrastructure that can run Node.js (parse-server - npm, 2021).	JavaScript	5 years	2,869
Sequelize	A promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite and Microsoft SQL Server (sequelize - npm, 2021).	JavaScript, TypeScript	11 years	3,359
Node-red	A framework that provides a browser-based editor that makes it easy to write APIs and online services (node-red - npm, 2021).	JavaScript	8 years	1,140
Strapi	A fully customizable open-source software that provides a headless content management system (strapi - npm, 2021).	JavaScript	6 years	2,878
Infor-design	A framework-independent UI library consisting of CSS and JS that provides tools to create user experiences (infor-design, 2021).	JavaScript, TypeScript, HTML, CSS	5 years	2,524
Electron	A framework that helps build cross-platform desktop apps with JavaScript, HTML, and CSS (electron - npm, 2021).	JavaScript, TypeScript, C++, Python, HTML	8 years	11,794
React	A library for building user interfaces (react - npm, 2021).	JavaScript, TypeScript, C++, CSS, HTML	8 years	9,673
Uglify-js	A parser, minifier, compressor and beautifier toolkit (uglify-js - npm, 2021).	JavaScript, HTML, Shell	9 years	1,197

have performed similar work (Paul et al., 2021; Bosu, 2014; Ebert et al., 2019); the number of analyzed projects in these studies varies between one to ten projects. We use the aforementioned Recent Activity metric as a basis to rank the projects, since such projects tend to be rich in pull requests, and consequently have higher chances for finding and discussing security issues (Walden, 2020).

Table 1 depicts the selected ten projects for our study. For each project, we present the project's domain, the programming languages used through the development lifetime, the age, and the total number of PRs in the project. As shown in the table, these projects cover multiple languages and application domains. Furthermore, the projects have a considerable long development lifespan and most projects have thousands of PRs in the project repository.

2.2. Identification of PR candidates

The goal of this phase is to identify PRs with security-related reviews in the selected projects. To that aim, we conduct a three-step methodology: (1) we elicit a set of security-related keywords, (2) we refine this list of keywords, and (3) we use the refined list to identify the set of PRs with security-related reviews. In the following, we describe each step in details.

(1) Obtaining initial dataset of security-related keywords. To identify relevant PRs that are of interest to our study, we use security-related keywords. Influenced by the related literature, we initially adopt a dataset of 48 security-related keywords used in the previous studies (Paul et al., 2021; Bosu, 2014). We use this initial set of keywords and apply each of them, using a regular expression, to identify security-related reviews in the projects. To this aim, we collect all the comments and discussions from all PRs in the projects and identify PRs that contain any keyword from our previously selected keywords dataset. After running the 48 keywords against all the PRs (i.e., 38,004 PRs), we identify 3503 PRs as candidates for our study.

(2) Refining the list of keywords. We identify 3503 candidates of PRs, however, through a preliminary manual inspection of the identified PRs, we observe that a considerable share of PRs were not relevant for our study, i.e., they do not discuss security-related issues. This is because some of the used keywords are specific to languages other than JavaScript, and hence, include a lot of noise in our PRs dataset. For example, the initial list of keywords contained keywords such as "css", which in many of

Table 2
List of refined security-related keywords.

Vulnerability type	CWE ID	Keywords
Race condition	362–368	Race, racy
Buffer overflow	120–127	Overflow
Integer overflow	190, 191, 680	Overflow, underflow
Improper access	22, 264, 269, 276, 281–290	Unauthenticated, gain access, permission
Cross Site Scripting (XSS)	79–87	Cross site, XSS
Denial of service (DoS)/Crash	248, 400–406, 754, 755	Denial service, DOS, *denial of service*[a], *DDOS*[a], *redos*[a]
Deadlock	833	Deadlock
SQL injection	89	Injection
Cross site request forgery	352	Cross site, CSRF, forged
Common keywords	–	Security, vulnerability, vulnerable, overrun, exploit, insecure, breach, threat

[a]Keywords in *italic* are our additions to this list.

the cases for our selected projects, e.g., Improve ctm edit (2021), 2834 - Adjust placement (2021) and Robust animation (2021), refers to cascading style sheets, the styling language for web pages, and not Cross Site Scripting (CSS) vulnerabilities, as it is commonly referred to in the security domain.

Therefore, to reduce the number of irrelevant PRs from our candidate set, we further verify and refine the list of keywords. First, we randomly select a statistically significant sample of the candidate set for each keyword. The size of the chosen sample is significant enough to satisfy a 95% confidence level with a 5% confidence interval for each designated population. Once the sample of PRs for each keyword is selected, we split the full dataset of identified PRs into two along each keyword (i.e., each dataset has half the number of PRs for each keyword), and a single author was assigned to each half of the dataset to review whether each PR could contain a discussion about security (i.e., the first two authors independently examined the PRs). Since this step was meant to eliminate keywords that produce too many false positives, the authors were encouraged to accept any PR that had even the slightest hint of a potential security issue, so that the most potentially useful keywords are maintained. Furthermore, to mitigate potential personal biases, the authors cross-referenced the number of PRs retrieved from each keyword to make sure that no author was dismissing PRs too easily and resulting in a lower amount of PRs filtered. If the keyword was used in a security discussion of the PR, then we consider the PR a relevant case at this stage of the study and irrelevant otherwise.

After going through all the samples of the PRs for each keyword, we decide the inclusion of the keyword by setting a reasonable threshold, i.e., if a keyword retrieves less than 10% relevant cases in its sample, we remove it from our list of keywords. Otherwise, the keyword is included. We chose 10% because we wanted to use a low threshold to preserve as large a variety of keywords as possible, as limiting the amount of keywords could bias the dataset towards only finding issues related to those specific keywords. This process yields 23 *refined keywords*. Note that as we go through the PRs of the relevant cases, we identified keywords that were being used in security-related reviews and were not in our initial list of keywords. This step ends us with three newly added keywords, namely, denial of service, DDOS, redos. Table 2 shows the list of the final 26 unique keywords used in our study. The keywords are associated with a Common Weakness Enumeration (CWE) (CWE, 2021). CWEs are well-defined classifications for the explored software weaknesses (e.g., CWE-121 corresponds to Buffer Overflow security vulnerabilities).

(3) Identifying PR candidates for our study.

We then use the refined list of 26 keywords and use an automated script to help search for the keywords in the comments of PRs. The PRs that have the keywords in their comments are added to our dataset of PRs for the study. This process yields a total of 882 PRs as candidates for our study.

2.3. Manual validation of the identified PR candidates

In the previous phase, we were able to curate a refined list of keywords to identify PR candidates (882 PRs), and reduce the number of irrelevant PRs for our study. However, some of those PRs might still be wrongly identified due to the limitation of our keyword search technique. For example, our technique flags this PR (Reload grant, 2021) as relevant because one of the review comments in the PR contains the "permission" keyword:

> "*...when users change the provider config from UI (Roles and Permission page), we will save the new config into db, and also sync it into JSON file as well, right?*"

As shown in the quote, the "permission" keyword is used in the context of a webpage name, and has no security implications to the project. Examples like this motivated us to further manually validate the set of 882 PRs. Hence, in this step we conduct a thorough manual analysis on the 882 PRs to filter out the irrelevant cases from our further analysis.

To filter out PRs that did not discuss security-related reviews (from our candidate set), two authors independently and manually look through all the 882 PRs. Both authors examine whether the contributors and reviewers of each PR discuss security-related topics in the PRs' comments. If so, the PR is included in the dataset for our later analysis. To evaluate the agreement between the two authors, we used Cohen's kappa coefficient (Cohen, 1960), which is a well-known statistical method that evaluates the inter-rater agreement level for categorical scales. In our manual labeling of the PRs, the level of agreement between the two authors was of +0.92, which is considered to be an excellent agreement (Fleiss and Cohen, 1973). At a macro scale, the authors extract the PRs information in eight rounds (almost 110 PRs per batch). Upon completion of each round, they meet and discuss any conflicts about including the PR. The goal of these meetings is to address any inconsistencies and to work together to resolve them. In the case that the authors have derived conflicting information about the PR, they will discuss until an agreement is reached.

Our in-depth manual analysis identifies 171 validated PRs with security-related reviews (out of 882 PRs), which span across all the projects in our dataset (Alfadel, 2023). Fig. 2 shows an example of a relevant PR that discusses a security issue during code review. In this figure, one of the PR's reviewers in the Marked project asked a second reviewer to check whether the code is vulnerable against a ReDOS security issue.

Finally, during our analysis we observe that the security issues discussed in the PRs could occur not only due to security concerns in the proposed changes in the PR but also due to concerns already existing in the system. Therefore, we also examine whether the PR is introducing a security issue or if the issue was already in the project/system prior to the PR. We find that 70.76% of the security issues in our dataset are raised by reviewers to

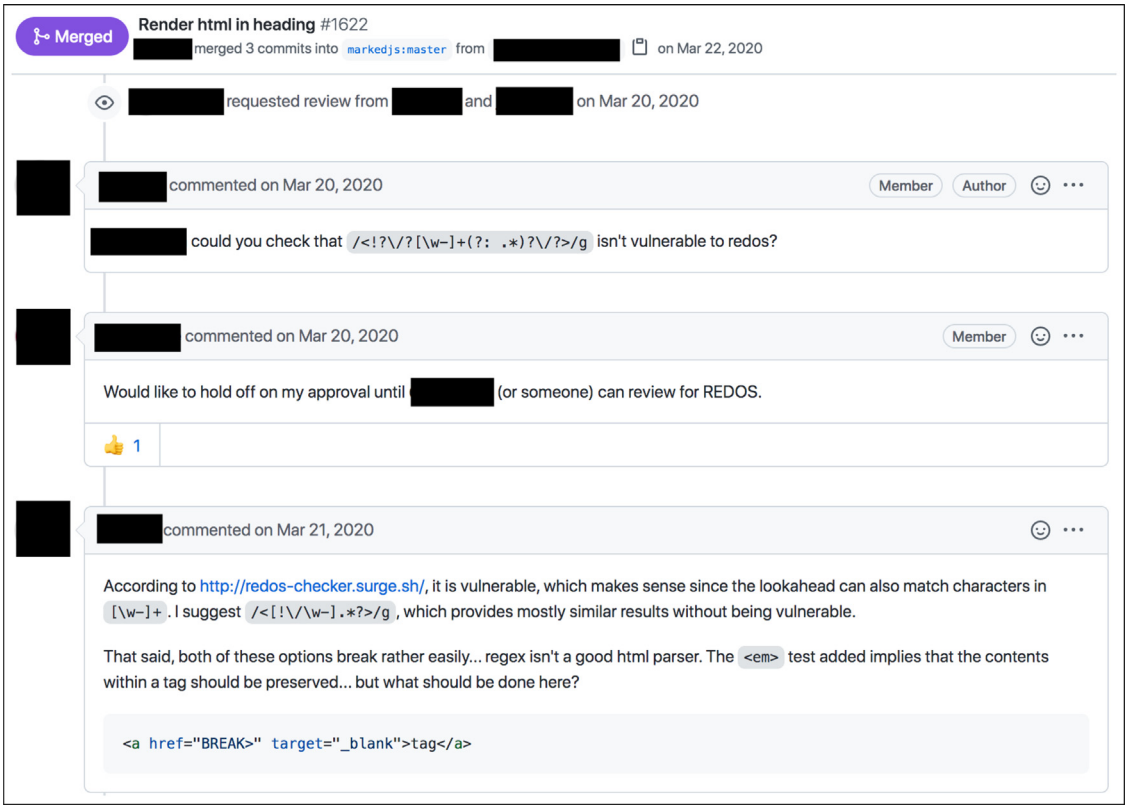

Fig. 2. Example of a security issue raised during code review (Render html, 2021).

point out a security concern related to the proposed PR. In other words, the changes made in the PR introduce security issues. For example, in Strapi PR,[1] the PR proposed a functionality related to database connection errors. However, reviewers raised concerns about exposing critical user information by logging information to the console. Stapi issue[2] is another example where the PR author proposed functionality that introduced a security concern raised by reviewers in the discussion. In 29.24% of the cases, we find that the PR is fixing security issues that were in the system before the PR creation time. For example, in this PR,[3] the author proposed a way to implement fresh tokens for the authentication flow, i.e., the functionality adds artifacts that allow application systems to perform the authorization and authentication process (which are security components in the project). During the PR discussion, we observe that reviewers were concerned about how the authentication is implemented, i.e., the refresh tokens are not secure enough because the proposed flow cannot verify which user is requesting the token.

3. Results

In this section, we answer our RQs. For each RQ, we provide a motivation, describe the approach, and present the results.

3.1. RQ1: How often are security issues identified during code review?

Motivation: The goal of this RQ is to gain an initial overview of the prevalence of security issues identified in a code review, allowing us to quantify the degree of effectiveness of code review for security purposes. In turn, this will help project maintainers and the community to set realistic expectations on the effort that is put in security-related code reviews and how often security issues are raised in regular code review processes.

In particular, we examine how often a security-related concern is raised in code reviews under three different granularity levels: (a) number of PRs; (b) number of files touched by the PRs; and (c) number of security-related comments in the PRs.

Approach: We employ the methodology explained in Section 2.3 to manually identify the PRs with security-related reviews in each of the studied projects. To identify the files changed in the PRs and the prevalence of security comments in the overall PR discussion, we use the metadata of each PR, to get the files that contain the security issue and the comments that discuss the security issue. More specifically, we use the following methodology to gather our results:

- To find the distribution of security issues at the PR level, we quantify the number of security-related PRs per project.
- At the file level, we quantify the number of unique files that contain the identified security issue per PR, and sum up the number of unique files for each project.
- Finally, at the comment level, we quantify the number of comments that specifically discuss the security issue per PR, and then sum up the total number of such comments for each project.

We normalize the result by the number of total PRs in the project, total files that the project has, and total comments in the analyzed PRs, respectively.

Result: Table 3 shows the distribution of the 171 PRs identified in the studied projects at different levels of granularity. We observe that the `Infor Design` project is the only project that had no

1 https://github.com/strapi/strapi/pull/3163.
2 https://github.com/strapi/strapi/pull/5330.
3 https://github.com/strapi/strapi/pull/2704.

Table 3
Distribution of security-related issues at different granularities, per project.

Granularity	Project	# Total	# Security-Related	%
PRs	Marked	758	27	3.56
	Moment	1,812	2	0.11
	Parse-server	2,869	19	0.66
	Sequelize	3,359	19	0.57
	Node-red	1,140	6	0.53
	Strapi	2,878	18	0.63
	Electron	11,794	47	0.39
	React	9,673	31	0.32
	Uglify-js	1,197	2	0.17
Files	Marked	327	9	2.75
	Moment	786	2	0.25
	Parse-server	413	15	3.63
	Sequelize	491	13	2.65
	Node-red	1,187	6	0.50
	Strapi	3,360	15	0.45
	Electron	2,130	42	1.97
	React	2,105	30	1.43
	Uglify-js	276	2	0.72
Comments in Security-PRs	Marked	665	108	16.24
	Moment	25	7	28
	Parse-server	380	49	12.89
	Sequelize	809	39	4.82
	Node-red	65	8	12.31
	Strapi	309	39	12.62
	Electron	1,250	137	10.96
	React	777	63	8.11
	Uglify-js	35	5	14.29

security issues raised during code review, i.e., the 171 cases are distributed across the remaining 9 projects in our dataset.

As a result, in terms of the number of security-related PRs, we find that the 171 PRs correspond to 0.49% (less than 1%) of all PRs in the projects. Concretely, the project `Electron` has the largest occurrence of security issues raised during code review (47 PRs), while there are only 2 PRs with raised security concerns in `Moment` and `Uglify-js` projects. Overall, the rate of PRs with security-related reviews varies from 3.56% in `Marked` to only 0.11% in `Moment`, showing that only a minority of PRs raise any concerns about security. Consequently, the identified security issues are concentrated on a small fraction of the projects' files (0.25%–3.63%). For example, marked has 27 security issues that are identified in 9 files out of the 327 the project currently contains.

Although the PRs with security-related reviews seem rare at both the PR and file granularity levels, once a maintainer expresses a security concern on the PR, maintainers discuss it at length in the PRs. Between 4.82%–28% of all comments in the 171 PRs are related specifically to the security related concern.

> Security issues are raised in a small fraction of PRs (0.11%–3.56%), affecting also a small fraction of project files (0.25%–3.63%). However, raised security issues are discussed at length by project maintainers (4.82%–28% of all comments in the PRs are security-related comments).

3.2. RQ2: What types of security issues are identified during code review?

Motivation: The goal of this RQ is to understand the types of security issues that project maintainers discover and discuss during the code review process. This investigation is important to help researchers and practitioners compare and contrast these results with other approaches, such as bug bounties, code inspections, all the way to software inspection tools that have been very well studied (e.g., Aloraini et al. (2019), Yang et al. (2019) and Imtiaz et al. (2021)). More important, such comparison will help us understand the types of issues where code review can be effectively employed and where maintainers may need better assistance to identify issues at code review time.

Approach: To categorize the identified security issues, we resort to in-depth manual analysis of the 171 PRs in our dataset. In particular, we follow the steps below.

Independent labeling. We start the manual classification of the 171 PRs. Initially, using the negotiated agreement technique (Mirhosseini and Parnin, 2017), we inspect the first 50 of the PRs together while the rest were inspected independently. The first two paper authors were assigned to engage in the manual card sorting of PRs. Each of the two paper authors read the content of the PRs and assigned a category without any external influences from the other authors. To analyze each PR at a granular level, first, we examined the title and body description of each PR. Furthermore, we inspect the review comments and discussion and the PR commits. Labels are created and assigned while inspecting the PRs, and every new label is discussed among coders and, if necessary, retroactively applied to previously labeled PRs. Note that the annotators attribute a single type of issue per PR, utilizing online search for the issue type in public advisories databases, e.g., the CWE classification (CWE, 2021), to help us classify the issue according to the CWE specification.

Compute agreement and solve conflicts. Once we conclude the independent classifications, we computed the Cohen's Kappa agreement coefficient (Cohen, 1960) for the two rater's classifications. We obtained 79% of agreement, i.e., substantial. We then held a meeting to discuss the disagreement instances one by one, so we could reach a consensus on the final labels for each issue.

Build taxonomy. After our final decision about the categories assigned to all 171 PRs under analysis, we hold a meeting to build the taxonomy of issue types of security-related review. In the meeting, we take inputs from the third author to finalize the taxonomy. During the meeting, we create the abstraction level of the taxonomy.

Result: Through our manual analysis, we find 14 different types of security issues identified in the PRs during code review. Table 4 shows the description and the frequency of each type represented in our dataset, sorted by frequency in descending order.

As seen from Table 4, some types are common across projects, i.e., they exist in a high number of projects. Others are more frequent within certain projects. Next, we explain some types in more details.

Common security types across the projects. From Table 4, we observe that 7 types are common across the projects, i.e., they exist in 3 or more projects.

For example, we find **23 Race Condition** security issues, which affect three projects in our dataset, namely, `Electron`, `Parse-Server` and `Sequelize`. A race condition can occur when two threads try to access the same data, which results in having wrong data in the threads or causing errors as a result of trying to use the same resource simultaneously with both threads. A race condition can compromise the application security when it causes threads to execute code in unintended ways. In the studied projects, this can happen in two ways. For example, in the Sequelize project, this occurs when two simultaneous callbacks within the JavaScript are racing to execute some functionality. In the other, and more common method in our dataset, a race condition can occur when threads are created in the C++ layer of the JavaScript project. This second method is seen most frequently in the Electron project, since it re-uses a lot of the Chromium code, which is a C/C++ based project, to render the user-interface (UI).

Table 4
Types of security issues identified during code review and their frequency.

Type	Description (with example)	Frequency	# Projects
Race condition	Occurs when two or more threads can access shared data and they try to change it at the same time, which may lead to multiple issues, e.g., alter, manipulate, steal data, and malicious code. An example of race condition in our dataset can be found here (protocol, 2023).	23	3
Access control	A system that does not restrict or incorrectly restricts access to a resource from an unauthorized actor suffer from Access Control security issue. An example of access control in our dataset can be found here (Allow read, 2023).	23	6
ReDOS	A regular expression denial of service (ReDoS) is an attack that produces a denial of service by providing a regular expression that takes a very long time to evaluate, which may lead to either slowing down the system or becoming unresponsive. An example of ReDOS in our dataset can be found here (Fix ReDOS, 2023).	23	1
XSS	XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. An example of XSS in our dataset can be found here (Changelog, 2023).	22	3
SQL injection	SQL injection attack consists of insertion or injection of a SQL query via the input data from the client to the application to spoof identity. An example of SQL Injection in our dataset can be found here (Adding support, 2023).	14	2
Documentation	In such cases, developers discuss issues related to enriching the project documentation for a better and more secure use of the project. An example of documentation in our dataset can be found here (docs, 2023).	14	5
Improper authentication	A weakness that allows an attacker to either capture or bypass the authentication methods that are used by a web application. An example of improper authentication in our dataset can be found here (Structured, 2023).	13	2
Sensitive data exposure	Occurs as a result of not adequately protecting a database where information is stored. An example of sensitive data exposure in our dataset can be found here (Add doc, 2023).	10	5
Remote code injection	Occurs when an attacker has the ability to run system commands remotely on the vulnerable application. An example of remote code injection in our dataset can be found here (fix, 2023).	9	4
Overflow	Occurs when the entered data in a buffer overflows its capacity to adjacent memory location causing the program to crash. An example of overflows in our dataset can be found here (feat, 2023).	7	5
Deadlock	Occurs when the software contains multiple threads or executable segments that are waiting for each other to release a necessary lock, resulting in deadlock. An example of deadlock in our dataset can be found here (Update Travis, 2023).	4	2
Improper input validation	Occurs the project does not validate the input properties that are required to process the data safely and correctly. An example of improper input validation in our dataset can be found here (Update Travis, 2023)	4	1
Vulnerable package	A third-party vulnerability contains a vulnerability. An example of vulnerable packages in our dataset can be found here (fix(postgres), 2023)	3	2
DOS	Occurs when an attacker floods the target application with traffic or sending it information that triggers a crash. An example of DOS in our dataset can be found here (Add reset, 2023)	2	1

An example of a race condition can be found in a CVE[4] of the Chromium project, for which the Electron package re-uses the code for. In this example, multiple audio contexts, using multiple rendering threads may call the same method that calls free on a specific memory location. Since this is not thread-safe, multiple calls of the free function on the same memory space can have undefined behavior, which could allow an attacker to exploit the corruption in a heap. According to OWASP,[5] the implications of such a vulnerability can range from crashing programs, altering the execution flow or as severe as allowing the attacker to execute code, usually with elevated privileges.

Access Control (23) is another security issue type that seems to be common across the studied projects; we find 23 cases distributed across 6 projects. The reason for this common security concern is that most projects have to manage several permission options, which may result in issues related to Permission and Access Control. For example, the `Strapi` project manages various permissions, and has encountered an issue in this PR (Fix deep, 2021). The proposed PR had vulnerable changes, allowing certain users to access files beyond their permission scheme, allowing to register themselves as admins when they should not be able to Unable to override (2021) and Add optional (2021).

In addition, our manual analysis finds several cases of **Sensitive Data Exposure (10)** issue, affecting 5 projects. We found that issues related to Sensitive Data Exposure can range from storing information in plaintext, logging sensitive information or even exceptions and error messages (e.g., Messaging API (2021), Add Connection (2021a) and Merge pull (2021)). For example, in some cases, maintainers raised the issue that the system was logging sensitive information that should not be exposed. As an example, in this PR (Add Connection, 2021a), the session id is used as a token generated on the server, and stored on the client by means of a cookie, which is used in later communications to identify the user. Developers may log the session id to track the user interactions with the application during a session, however, if an attacker gets access to live logs, she could use the session id to impersonate active users.

Cross-site scripting (22) is another common issue, which is triggered at the client-side when a potential attacker sends malicious code in the form of a browser side script, e.g., to hijack user's account and credentials. In our analysis, we find that the Cross-site scripting (XSS) type affects three web-based

[4] https://www.cve.org/CVERecord?id=CVE-2021-30603.

[5] https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory.

projects in our dataset. We find 22 cases of XSS issues where a lot of HTML components are rendered in the projects. A lot of these issues arise when the project is failing to properly escape the inserted HTML, which can cause unwanted cross-site scripting attacks. As a result, many developers spend a lot of time discussing and trying to figure out the best approach to escape potentially malicious HTML, such as in the case of this PR (escapeTextContentForBrowser, 2021).

Other commonly discussed security types include Overflow, Remote Code Injection and Documentation. In fact, we find the **Documentation (14)** cases to be important; they act as the communication medium between package maintainers and application developers who use the package. In such cases, package maintainers discuss inadequate or incomplete documentation of security critical usages of the package functionality. For example, in this PR (docs, 2021), a project maintainer states the following:

> "*..it can lead people to the incorrect assumption that they should actually run a server in the main process, which will confuse folks with this comparison. The relationship between browser/renderer process has fundamentally different security, performance and communication constraints than a traditional client-server model that cannot be sufficiently explaining in this comparison*".

As shown, the case specifies that the proposed documentation fails to properly explain the security constraints of the system, which can mislead package users. Documentation cases can mainly occur in PRs in two ways. The first is in changes that update the documentation, but none of the source code of the project. This type of documentation either will be published to the package's main webpage, in the README, or in some other form of the project chooses. The second case is changes that directly touch the source code. This method can affect how the exposed API, functions and comments are written which could mislead users of the package into using it insecurely

Other cases are related to attributes misnaming. Such issues indicate, for example, misnaming a variable to make it seem more secure than it actually is. An example of this case can be seen in this PR (Improve soundness, 2021a), where a variable is called SafeValue, which has security connotations, while it actually is not necessarily secure (Improve soundness, 2021b). Such a case can cause users of the package to misuse its API and introduce vulnerabilities into their own applications.

Frequent security types within specific projects. From Table 4, we also observe that some types are more frequent in specific projects. For example, we find **23 ReDOS** cases. ReDOS is one form of denial of service attacks, which occurs by providing a regular expression that takes long time to process, which causes a system to crash or take a disproportional amount of time. We find ReDOS issues affecting only the `Marked` project, a compiler for parsing markdown formats. When parsing markdown text, `Marked` uses a lot of regular expressions, and as such, is very prone to creating such issues, as shown in various PRs (fix image, 2021a; Fix GFM, 2021a; enable, 2021). In fact, we observe that the project maintainers of Marked seem to be employing a static analysis tool (called *vuln-regex-detector davisjam, 2021*), integrating the tool into their CI pipeline, as shown in this PR (test, 2021). Moreover, the maintainers seem to seek the help of someone in the team who is regarded as a security expert to help detect whether certain regular expressions are actually vulnerable. We observed his involvement in most ReDOS issues (e.g., security (2021a), Fix GFM (2021a), enable (2021) and fix image (2021b)). This indicates that such issues of ReDOS are not straightforward to spot during code review, and may require automated tools as well as "security experts" to better identify and validate the cases.

Similarly, we find several cases of **SQL injection (14)**, which affect two projects only, `Sequelize` and `Strapi`. SQL injection issues generally produce attacks by injecting SQL queries via the input data to spoof identity. We observe that 10 out of 14 cases are present in `Sequelize`, which is a powerful package in the JavaScript, which deals with managing SQL databases, and hence, it is more susceptible to SQL injection issues. Most of SQL injection cases that need to be dealt with in our dataset are cases that require escaping characters from potentially malicious strings that would inject SQL queries that grant unauthorized access to the database (6935 remove, 2021; Don't quote, 2021a).

Issues related to **Authentication (13)** are mostly present in the `Parse-Server` project (6/12 cases), where user authentication is managed. In such cases where the authentication is not properly managed, a potential attacker would gain access to sensitive data or functionality. For example, in this PR (Structured, 2021a), a reviewer noticed that a public unauthenticated request was returning information about the server version. This can be risky as attackers are allowed to send requests to servers to check if they are running a vulnerable version of the server and take advantage to perform further exploits.

We find other less frequent types, e.g., Deadlock, DOS, Improper Input Validation, and Vulnerable Packages. An example of a security issue related to deadlocks can be seen in a PR for the Sequelize project.[6] We find that the developers are using a PR to discuss several issues, one of which being a deadlock issue in their code that interacts with the PostgreSQL database. Since sequelize is a package that provides object-relational mappings between JavaScript objects and database objects, it is very likely that a package user would use this package on a server and expose some sort of interface (such as a REST API or web page) for a client to consume. The package user, through no fault of their own, could accidentally expose this deadlock through their interface in a reproducible way, and if that occurs, a malicious actor could easily abuse the application interface to cause deadlocks in their server-side application. This would inevitably affect the hosting server(s)' ability to continue to function, and ultimately hang and would render the service unusable, much in the same way a denial of service attack would.

Finally, note that we compare the types of security issues identified during code review with the post-release security issues (i.e., advisories) in the Discussion Section. Security advisories are security vulnerabilities that affect the post-release version of the projects and have been announced and reported in public databases.

> Our investigation showed 14 types of security issues raised in code review. Issue type varies from common issues found across the projects, e.g., Race Condition, Access Control, XSS, Documentation, and Overflow, to other types more frequently affecting specific projects, e.g., ReDOS, SQL injection, and Authentication.

3.3. RQ3: How do developers respond to the identified security issues during code review?

Motivation: As shown in RQ_2, various types of security-related issues are brought up during code review, however, how the issues are tackled, if at all, is crucial to understand the effectiveness of the code reviewing process. Therefore, in this RQ, we investigate how developers respond to the identified security issues and the mitigation strategies employed. Doing so is important to

[6] https://github.com/sequelize/sequelize/pull/6443.

Table 5
Response themes for handling the 171 identified security issues.

Response theme	Description	# Total (%)
Fixed	The security issue is raised during code review and evidence for a related fix is observed.	79 (46.19%)
No threat	Project maintainers come to a conclusion that the security issue poses no real threats to the project.	49 (28.65%)
PR rejected	The raised security issue caused the PR to be rejected by project maintainers.	15 (8.77%)
Not fixed	Project maintainers opted not to fix the raised security issue, often due to very complex technical difficulties.	14 (8.18%)
No response	The security issue is raised during code review, but no discussion and changes are made to reflect on these concerns.	8 (4.67%)
Not fixed and discuss general issue	Project maintainers discuss a security issue that is not directly relevant to the reviewed PR but rather a general security concern to the project.	6 (3.54%)

motivate improvements of the code review process, with the aim of increasing its effectiveness.

Approach: To find out how developers respond to the identified security issues in our dataset, we manually inspect review comments associated with the 171 PRs in the dataset. In particular, we examine whether the security issue is resolved in a way that increases the overall security of the related project, and how the issue was tackled and mitigated during the discussion. Similarly to RQ_2, two authors independently classify the responses using an open card-sort method (Fincher and Tenenberg, 2005), where labels are created during the labeling process, by looking through the discussions, code changes and commit history that occurred through the code review process. For this manual labeling, a high level of agreement is reported with Cohen's kappa coefficient of +0.93. Once again, when different labels were assigned to the same PR, the annotators discuss them to reach a consensus.

Result: Table 5 presents the themes of the responses to the issues raised in the 171 PRs, identified by our manual analysis. Below, we provide more details about each response theme.

Fixed (46.19% of cases). This is the most common way of responding to the identified security issues. In such cases, we find that the security concern was discovered during code review in addition to an evidence for a related fix. In most cases, we observe that the issue is fixed, and the PR is merged (e.g., Add Connection (2021b)). In other cases, we observe that the security issue is fixed, but the code is flawed for some other non-security reason, which led the PR to be closed (e.g., Roles (2021)). In a few cases (e.g., Don't quote (2021b)), we observe that the reviewers suggest to open a new PR to properly design the solution that tackles the raised security issue. An example of a fix is given in this PR (Structured, 2021a). In this example, the maintainers found an authentication issue where a potential attacker could have access to a public unauthenticated request, as stated:

> *"... Returning information about the server version on a public unauthenticated request makes it really easy to develop bots that check for a version of Parse Server that is vulnerable to an attack, lowering the cost of effort for a random attacker to locate vulnerable servers".*

The same maintainer also suggests a solution for the issue:

> *"... If the Health Check is going to return structured data, I think that's a feature that should be possible to disable for security hardening. I'd prefer to see the health check do more - but still just return OK. Specifically, it would be nice if it did a simple round-trip to the database that does nothing but confirm the database server is up..."*

Other maintainers agreed on the relevance of the issue and the suggested fix as well. Hence, the issue was fixed by removing the related information of the version in the JSON response, as shown in this commit (Structured, 2021b).

No Threat (28.65% of cases). Of the total number of the analyzed security issues, we observe that 28.65% of the cases do not impact the corresponding part in the project. In such cases, project maintainers did not reach a consensus on the identified security issue. We observe that a reviewer pointed out a security issue, but was effectively deemed by other reviewers and/or by the contributor that it was actually not a security concern, i.e., the identified issue was not a real threat to the project. For example, as shown in this PR (Protected, 2021), a reviewer discussed some flaws related to the design of the permission feature and how it works. However, after the discussion, the reviewers agreed that it did not seem to be of any security concern. One maintainer stated:

> *"... the pattern of code was actually widely used in the project and known to not have security concerns".*

In other examples where the raised issue is not considered as harmful (security, 2021b), project maintainers find that the issue cannot be triggered accidentally in a normal context, i.e., the package users should know how to use the functionality in a secure manner:

> *"...we should educate our dependents on the safe way to deal with parsing user input. (i.e. web worker/vm. runInNewContext)".*

In other cases, project maintainers find that the identified security issue has no direct impact on the project. For example, in this PR (Update, 2021) that concerns about a vulnerable dependency, the vulnerability does not affect the end users, since the dependency is solely used as a development dependency:

> *"... for users who are using marked, they do not see (and are not affected) by dev dependency vulnerabilities".*

PR Rejected (8.77% of cases). In 15 (8.77%) PRs, the raised security issues caused the PR to be rejected by the respective project maintainers. For example, in those PRs (e.g., Feature (2021), Add information (2021) and Fix deep Me (2021)), the proposed changes in the PR are vulnerable, and not easy to fix. Given that the proposed changes in the PR are discussed to be lower on the priority list, the team decided to close the PR to avoid the raised security concern. In this example Fix deep Me (2021), the maintainer stated:

> *"Closing this PR because of security issue... With this, we can access to all the users base of a group".*

Not Fixed (8.18% of cases). We observe that in 14 PRs, the project maintainers opted for not fixing the raised security issue, often due to very complex technical difficulties. For example, in this PR (Add CounterCache, 2021), the project maintainers clearly discuss a race condition in the code. However, through our manual inspection, we observe no action was taken in the PR to fix the race condition since the maintainers do not seem to be able to find where the issue is coming from or do not seem willing to invest time into it at this point of the project, as stated:

> *"...imperfection is to be expected when there is only been one iteration :)"*

Another example (e.g., Changelog (2021)) is where project maintainers decided not to fix the issue in the current project release, as it would cause a breaking change and the fix could require significant code changes that replace the entire underlying requirements of the PR. In this case, the project maintainers prioritize respecting the release deadlines over fixing the security issue.

In some other cases, the project maintainers are offloading the responsibility of security to its users (the dependent applications). In this example Block non (2021), the maintainers stated:

> *"...there is a responsibility up to the developers of Electron projects to ensure the content they are pulling in is safe and trusted".* And that *"Electron intentionally breaks security and the sandbox to make applications possible".*

This indicates that in certain cases, the project maintainers need to weigh the pros and cons of securing their project, as there is a tradeoff between usability for the project users and security of the project itself.

No Response (4.67% of cases). The cases under this category are concerns that are raised by maintainers but were completely ignored within the context of the PR. For example, in these PRs (Fix GFM, 2021b; Migrate, 2021), we observe that a specific reviewer raised an issue related to ReDOS, but we could not find any evidence of a discussion or any response back within the comments of the PR or any of the commits and discussions that referenced the related PR. Similarly, in this example Fix GFM (2021b), one maintainer raised a potential issue related to ReDOS issue, and asked another maintainer to validate it. However, we did not observe any response back from other maintainers. In other cases, we observe that the security issue was not raised early enough, i.e., the issue was identified only after the PR's decision was already taken (closed/merged). As an example, in this PR (fix(redshift), 2021), we find that, after merging the PR, a developer adds a comment concerning a potential SQL injection. In such cases, we find no evidence of discussing or addressing the security issue after being raised.

Not Fixed and Discuss General Issue (3.54% of cases). In such cases, reviewers discuss issues that are not directly relevant to the reviewed PR, i.e., they discuss general issues that come along the discussion of other related issues. This can be some improvements for security-features or potential approaches to fix security issues. In such cases, no actions is taken in the PR since the discussion is not specific to the PR changes. For example, in this PR (Escape component, 2021), the project maintainers discuss various ideas and approaches to escaping characters to prevent potential XSS issues, though no actual XSS issue is raised. Throughout this discussion, they identify potential security issues in each others' ideas and refine them to come up with an optimal solution. This helps project maintainers plan out a secure approach to prevent a potential vulnerability from an undiscussed plan. As shown in the following quote:

> *"...I'd be curious to see if an indexOf('.') !== −1 check before escaping would help perf in the common case. Since the common case is no dot. We could also escape to a format that does not need re-escaping when it goes into the DOM attribute. Since we already have one escape pass, we can utilize that for both. Might be dangerous though. Easy to open up XSS vulnerabilities".*

The author is relating the content of a PR to some future work or feature, and discussing their security concerns for the possible approaches. This provides an entryway for a discussion to further discuss how they should handle these future works, before they begin tackling them.

The majority (54.96%) of the identified security issues during code review are fixed and mitigated. However, many of the issues seem to be considered as having low threats to the projects (28.65%). In a few cases, the project maintainers do not fix the issues (8.18%) or even respond to them (4.67%). Interestingly, some of the issues are not directly related to the reviewed PRs, but are still discussed during code review (3.54%).

4. Discussion & implications

In this section, we discuss our results further. First, we perform an analysis of code reviews in projects that are not in the advisories dataset (4.1). Then, we present a discussion on the comparison of security issues identified during code review to the post-release security issues (advisories) that are only identified after the project release (4.2). We also perform an analysis on the usage of tools as a solution to identifying security issues during code review (4.3). Finally, we provide insights about how our findings can improve the practice for researchers and practitioners (4.4).

4.1. Security code review for non-advisory projects

To assess the generalizability of our results to more typical projects that npm users would download, we perform the analysis of RQ_1 on a new dataset of 10 projects. These projects are selected based on our Popularity and Recent Activity criteria (in Section 2.1), but do not have any vulnerabilities reported in the advisories. This is to eliminate the potential biases associated with studying some of the most vulnerable projects.

We begin our analysis by searching for the most popular packages (using the number of downloads). We use the number of downloads to rank the projects by popularity, since the more popular packages would be more representative of a typical package that a npm user would download and use. Thus, we use the npm API to collect the more up-to-date data on the number of downloads for all the packages in the dataset. From this newly acquired data, we are able to select the top 10 most popular projects, whilst filtering out projects that do not meet our activity requirements or contain npm advisories. Once selected, we collect all comments from the pull requests on their GitHub projects. From there, we are able to identify which pull requests contain comments with security keywords, and we begin running our analysis of these pull requests to compare and contrast with the advisory projects. We follow the same approach applied in RQ_1 to conduct our analysis.

Table 6
Distribution of security-related issues at different granularities, per project (non-advisory projects).

Granularity	Project	# Total	# Security-Related	%
PRs	Babel	15,463	12	0.077
	DefinitelyTyped	6,288	22	0.34
	Regenerator	117	0	0
	DomUtils	1,202	0	0
	Entities	290	0	0
	Electron-to-Chromium	38	0	0
	TypeScript	1,005	4	0.39
	RxJS	374	3	0.80
	Core-JS	54,154	1	0.002
	caniuse-lite	3,740	0	0
Files	Babel	28,223	23	0.08
	DefinitelyTyped	61,698	47	0.07
	Regenerator	71	0	0
	DomUtils	33	0	0
	Entities	42	0	0
	Electron-to-Chromium	22	0	0
	TypeScript	65,708	4	0.006
	RxJS	1,330	2	0.15
	Core-JS	3,625	1	0.03
	caniuse-lite	843	0	0
Comments in Security-PRs	Babel	134	14	10.45
	DefinitelyTyped	302	24	7.95
	Regenerator	0	0	0
	DomUtils	0	0	0
	Entities	0	0	0
	Electron-to-Chromium	0	0	0
	TypeScript	22	5	22.73
	RxJS	42	4	9.52
	Core-JS	11	6	54.54
	caniuse-lite	0	0	0

Table 6 shows the distribution of the PRs identified in the studied projects at different levels of granularity. Concretely, the projects Babel and DefinitelyTyped have the largest occurrence of security issues raised during code review, i.e., 22, 12 respectively, while there are only 4, 2, and 1 PRs with raised security concerns in the remaining three projects. Overall, the rate of PRs with security-related reviews is quite low, showing that only a minority of PRs raise any concerns about security. Consequently, the identified security issues are concentrated on a small fraction of the projects' files ($<1\%$). Although the PRs with security-related reviews seem rare at both the PR and file granularity levels, once a maintainer expresses a security concern on the PR, maintainers discuss it at length in the PRs. Between 7.95%–54.54% of all comments in the 41 PRs are related specifically to the security-related concern.

Overall, both advisory and non-advisory projects follow a similar trend where security issues are raised in a small fraction of PRs, affecting also a small fraction of project files. Also, raised security issues are discussed at length by project maintainers in the PRs.

4.2. Comparison with advisories dataset

Code review identifies security issues before these issues are merged in the codebase and go to production. However, security issues uncaught by code review may later become known vulnerabilities in the projects. To better understand the effectiveness and limitations of code review, we compare issues identified during code review to post-release security vulnerabilities (advisories) that have been reported after the project release production. Such comparison will help us understand whether there are certain types of issues that code review can be effectively employed to identify.

To that aim, we resort to using the security advisories database provided by npm (npm, 2021). We collect all npm security advisories of the studied projects in the same timeline of the collected security-related PRs, i.e., we collect all advisories that have their publication date before August, 2021.

We report the results of our comparison by cross-referencing the security types identified during a code review with the types of advisories that affect the projects in our dataset. We manually check whether each one of the 14 types identified during code review (RQ_1) exist in the advisories dataset. Table 7 shows the types identified in our study, and whether they are mentioned in the advisories dataset. From the table, we can observe that four types in our study are not mentioned by the advisories dataset, namely *Race Condition*, *Access Control*, *Documentation*, and *Deadlock*.

Based on our observations, **the nature of the types that were more commonly found in code review requires in-depth knowledge of the project domain and implementation specifics.** For example, issues related to Race Condition and Deadlock stress this point as their identification require an in-depth understanding of the problematic code, how the concerned threads interact to deliver the desired functionality and only then can one begin to look for edge cases in which a race condition or a deadlock may arise.

Similarly, our observations seemed to hint that Access Control issues are difficult to spot without experience working with the project. In the case of Access Control (e.g., in this PR Add rate, 2021), we observe that in order to understand whether certain resources can be exposed or not, a deep understanding of the project users' requirements is necessary to understand whether those resources are sensitive and need special access to use. In fact, automated tools (e.g., static and dynamic testing tools) can help to detect the absence of access control in a system (Aloraini et al., 2019; A5:2017, 2021), but cannot determine whether it functions properly when in use. Contrary to this, code reviewers can take the time during the review to analyze whether the given changes are functioning as expected, which can help them identify whether the given functionality has an access control issue. Furthermore, we check if the security issue types (i.e., access control and race conditions) affect any npm package in a

Table 7
Cross-reference the types of security issues identified during code review with advisories dataset for the studied projects. The values in parentheses represent the number of affected projects.

Types in code review	Mentioned in advisories
Race condition (3)	–
ReDOS (1)	✓ (4)
Access control (6)	–
XSS (3)	✓ (3)
SQL injection (2)	✓ (1)
Documentation (5)	–
Improper authentication (2)	✓ (1)
Sensitive data exposure (5)	✓ (2)
Remote code injection (4)	✓ (3)
Overflow (5)	✓ (1)
Deadlock (2)	–
Improper input validation (1)	✓ (2)
Vulnerable package (2)	✓ (3)
DOS (1)	✓ (2)

recent version of the advisories dataset (collected from GitHub Advisories[7]), i.e., we do not limit the search to the advisories of the studied 10 projects in our dataset. We find 20 projects in the npm advisories where the project is affected by issues related to access control vulnerabilities.

Also, we check if the low number of issues related to access control is specific to JS. We perform similar analysis on other common package managers (i.e., Maven (Java) and PyPi (Python)). We find that the number of cases is 78 advisories (for Maven) and 40 advisories (for PyPi). Such results indicate that the number of access-control issues in package ecosystems is generally low, though this number is relatively lower in JS packages compared to other package managers like PyPi and Maven. In fact, prior work (e.g., Rennhard et al. (2022)) showed that detecting "logical vulnerabilities" such as access control is much more difficult than "technical vulnerabilities" such as XSS and SQL injection, as existing vulnerability scanners are not able to assess whether the access control rules are enforced correctly unless it has additional information about how the web application should work correctly. We believe this is one possible explanation behind our observation where we also find that the number of identified access control issues during code review (23 cases) is higher than the total number of access control advisories (20 cases), indicating that manual methods (e.g., manual source code analysis) are required to verify and identify such issues.

We perform a similar analysis for race condition issues. We find that the npm advisories contain a total of 4 cases, while there are 18 cases in Maven and 11 cases in PyPi. Similar to access control cases, we believe race condition issues are logical security issues where vulnerability scanners are not well-suited to detect them easily. Through our manual analysis of race condition cases, we observe that developers had a lengthy discussion to decide on the validity of the issue. In many cases, no action was taken in the PR to fix the race condition since the maintainers do not seem to be able to find where the issue is coming from or do not seem willing to invest time into it.

Issues classified as the "Documentation" type refer to security constraints on the project usage that are not properly communicated. Clients (i.e., project users) rely on documentation to understand how to properly use a project or its API. Any missing or unclear documentation may mislead these clients to use the project in unsafe manners, potentially leading to a vulnerability unbeknownst to them. Hence, properly written Documentation is extremely important due to the reliance from project users. Yet, such issues are not really exploitable vulnerabilities that may affect the project itself, and hence, the "Documentation" category is not present in the advisories dataset, despite being a major security concern for project users.

While some types of security issues are frequently identified through code reviews, we find that some other types are more frequently detected in the advisories dataset. For example, as seen in Table 7, we find that code reviews identified ReDOS in one project only (Marked). However, the advisories dataset mentions four projects (including the Marked project) affected by ReDOS, namely, `Moment`, `Uglify-js`, and `Sequelize`. We observe in RQ_2 that the project maintainers of `Marked` integrate a static analysis tool in the project pipeline and periodically invite a security expert to validate and fix specific issues like the ReDOS type, this was not observed for the 3 other projects. Such results may indicate that some other types of issues like ReDOS are easier to detect by means of tools or specialists.

Interestingly, we find that such observations are inline with a recent report by IBM company. According to IBM (What is threat hunting, 2021), around 80% of cyber-attacks could be handled by automated security tools. However, the remaining 20% of threats are more likely to include sophisticated threats and cause significant damage as on average they remain undetected for 280 days. The report also highlights that such bypassed threats require professionals and security analysts who understand the operations well, to be able to search, log, monitor and fix threats before they can cause serious problems.

In the case of Race Conditions, Deadlocks, or Access Control issues, this is inline with our assumption that project maintainers, who understand the underlying operations of the project, are better suited to find these types of issues, as they are well interconnected with the project functionality. This may help explain why Race Conditions, Deadlocks, and Access Control issues were only found by project maintainers during code review and not through the advisories. If Race Conditions, Deadlocks and Access Control issues require a deep understanding of the project to identify and mitigate, then it is less likely that people outside of the project maintainers would be able to find and report these types of vulnerabilities to the advisories.

Finally, we manually examine categories of security reports in the advisories dataset and count the frequency of each one. While our advisories dataset includes more than 600 vulnerabilities, we find that 78.27% of them are concentrated in 10 categories. Table 8 shows these 10 categories and the frequency of each one. Of the top 10 categories, we find 8 categories that were also identified in code reviews. Two categories (Prototype Pollution and Arbitrary Code Execution) were not identified in code reviews of the studied projects. Such results indicate that code reviews can identify common security advisories (e.g., XSS, SQL Injection, Authentication, etc.). However, there is still plenty of room for improvement. We observe that code reviews do not identify other common categories in advisories (e.g., Prototype Pollution). For example, from Table 8, we found that Prototype Pollution is the second most common category in the advisories, with 69 vulnerabilities affecting packages from different domains, including Parse-Serve and Node-Red packages in our dataset. With prototype pollution, an attacker can control the default values of an object's properties. Recent research has proposed techniques to detect prototype pollution vulnerability. For example, Li et al. (2021) proposed a static taint analysis tool to detect prototype pollution vulnerabilities in npm packages. They found 61 previously-unknown vulnerabilities.

Therefore, our results suggest that researchers should direct their efforts to improve practices and tools that tackle such vulnerability types during code review. This would significantly benefit a wide range of software projects to review their code against these categories. Moreover, package maintainers are encouraged to widely adopt such research tools in their code review process to constantly identify their vulnerabilities and fix them as soon as possible.

[7] https://github.com/advisories.

Table 8
Ranking of the top 10 vulnerability categories in the advisories dataset.

Category	Frequency
XSS	157
Prototype pollution	69
ReDoS	65
Command injection	64
Denial of service	59
SQL injection	28
Sensitive data exposure	22
Remote code execution	19
Arbitrary code execution	14
Improper authentication	11

4.3. Tool usage for security code review

Our results (RQ_2) show that in some cases, project maintainers integrate automated tools in the pipeline of the project development cycle. For example, we observed that the maintainers of the Marked project integrate a static analysis tool into their project's continuous integration pipeline. To report on the extent to which it was possible to detect the usage of tools as a solution to identifying security vulnerabilities, we manually investigate the discussion comments of the 171 security-related PRs and their relevant issues in search of any indicators or references of a security tool used in the studied project. We find that several projects adopt a variety of tools to protect the project from potential security issues. Our investigation leads us to the following observations:

- The project Marked uses a plugin called *vuln-regex-detector* (davisjam, 2021), which is a security tool that scans for ReDoS issues in the project. As seen in this PR,[8] the project configures the tool by defining it in the package.json.
- Another project (the Moment package) uses a tool called Debricked (GitHub, 2023), which is a security tool that analyses the latest commits and PRs of the project for known vulnerabilities. As shown in this PR,[9] the project integrated some changes to fix vulnerabilities affecting the project.
- Snyk tool (GitHub integration, 2023) was also used by several projects (e.g., Strapi and Parse-Server) to keep dependencies up to date in the projects. For example, in the project Parse-Server, Snyk has created this PR[10] to fix one or more vulnerable dependencies of the project. Another example is shown in the project Strapi.[11]
- Sonatype (IQ, 2023) is another tool adopted by several projects (e.g., UglifyJS and Sequelize). For example, as shown here,[12] Sonatype raises a vulnerability in the project UglifyJS. Other relevant examples can also be seen in these example.[13][14]
- Other observed tools include the NSP tool (nodesecurity, 2023). Project repositories rely on it to check if the package has been reported for security issues. For example, the project Marked discussed that the NSP security check tool reported that the package has a serious "Regular Expression Denial of Service" problem.[15]

[8] https://github.com/markedjs/marked/pull/1220.
[9] https://github.com/suculent/thinx-device-api/pull/400.
[10] https://github.com/parse-community/parse-server/pull/7825.
[11] https://github.com/strapi/strapi/pull/12314.
[12] https://github.com/mishoo/UglifyJS/issues/5721.
[13] https://github.com/sequelize/sequelize/issues/15172.
[14] https://github.com/mishoo/UglifyJS/issues/5699.
[15] https://github.com/markedjs/marked/issues/947.

Finally, note that it is possible that the list of observations above is not exhaustive. Still, we believe that adopting such tools can lead to identifying a wide range of security vulnerabilities. For example, Davis et al. (2018) studied the impact of ReDoS vulnerabilities in npm and PyPi. They found (using the tool vuln-regex-detector) that thousands of regexes are affecting over 10,000 modules across diverse application domains.

4.4. Implications

In this section, we provide some implications to practitioners and researchers.

Certain security issues are more commonly identified through code reviews. Our findings showed that a variety of security issues are identified during code review. However, we also found that certain types of security issues, e.g., issues relating to Race Conditions, Access Controls issues existed in the studied projects which were not frequently reported in the advisories dataset. Through our manual analysis (RQ_2 & Section 4), we discussed that dealing with such security issues is difficult and hard to locate. For example, in the case of Race Conditions and Deadlocks, the reason due to these issues being difficult to identify and fix is that they require a deep understanding of how the project uses multi-threading and what can cause Race Conditions. In the case of security issues related to Access Control, this is due to the need for a solid understanding of the project users' requirements, which is necessary to understand whether the accessed resources are sensitive and need special access to use. That said, our results show that code review is considered a critical approach for identifying specific security issues that require logical analysis of the issue.

Integrating automated tools in the project development cycle can help developers to identify critical vulnerabilities during code review. Our results (RQ_2) show that, in some cases, project maintainers integrate automated tools in the pipeline of the project development cycle. For example, we observed that the maintainers of the Marked project integrate a static analysis tool into their project's continuous integration pipeline (called *vuln-regex-detector davisjam, 2021*), which aided in the identification of most ReDOS security issues in the project. Also, through our manual analysis, we observed that, in several cases (e.g., nested parentheses (2021) and added data (2021)), the project enable tools for dependency management to upgrade outdated and vulnerable dependencies. For instance, in this PR (Add snyk, 2021) of the project Marked, several outdated dependencies were automatically updated and fixed by the Snyk tool. These results indicate that it is of great help for projects to use automated tools to target security issues that could affect the projects. Moreover, further research should explore different tools that can be integrated in the development cycle of projects to target security concerns. One particular example is CodeQL (github/codeql, 2021), which is a code analysis platform for finding zero-days and critical vulnerabilities in pull requests. While such tools have been proposed for a while, there is little known about the effectiveness of integrating such tools in software projects. Future research should examine the efficiency and effectiveness of such code review tools across projects for different types of security concerns. Such research is important to increase developers awareness to code review tools that can be employed in the project development cycle to help identify security concerns in their projects.

Overlooked security issues should be documented and reported to project users in an easily accessible way (i.e., project

READMEs, package descriptions, or package documentation). Although our findings show that identified security issues are frequently fixed, we found a non-negligible share (13%) of issues identified during code review end up not being fixed or are ignored by maintainers (see RQ3). We observed in several cases that such issues generally take great effort to mitigate or may contradict the goal of the project. In some cases, the maintainers state that the responsibility of the security issue is on the user to mitigate. However, in all of these cases, the project maintainers do not seem to come out of their discussions within their PRs with anything actionable that could help better advise users of such security issues This can impact the project users as proliferate to the project users unbeknownst to them. Therefore, we recommend to all project maintainers to document all potential security issues that could come about by using their project in a way that is easy to understand and easy to access by the users of their projects. In addition, project users cannot be expected to sift through the project history to gain a better understanding of what is their responsibility for security, and to understand what is not being handled by the project. Having some easily accessible documentation, such as in the README of a project, in the package description (like on the npm registry) or on the project website, can help give a high-level overview to prospective users of what they need to do to handle and mitigate such security concerns in their own applications.

Security issues raised in the studied PRs are raised on a small fraction of project files. Our results show that security issues identified during code review of the studied projects are very localized, only appearing in a small fraction of project PRs and files (see RQ_1). This indicates that project maintainers need to concentrate on these parts and pay more attention to them during security code review. Therefore, one way to support the code review process is to build tools that rank files based on their security sensitivity. For instance, files that have had security issues identified in them can be flagged by the tool as security sensitive. Such tools can help project maintainers for prioritizing code review for security issues, e.g., a PR that touches a file that has been flagged as security sensitive before may require the review of a security code expert.

5. Related work

There is plethora of work studying the effect of the code review process in finding defects. Thongtanunam et al. found that developers are often most concerned about documentation and structure to enhance evolvability, and fix functional issues (Thongtanunam et al., 2015). Beller et al. revealed that most changes of open-source systems in code review are indeed related to the functionality aspect (Beller et al., 2014). The study by Bacchelli and Bird (2013) showed that most changes of open-source systems in code review are also related to functionality. Mäntylä and Lassenius (2008) reported similar outcomes for other industrial and academic projects. McIntosh et al. (2014, 2016) examined the impact of code review coverage and participation on the code review quality. They found that projects with low code review coverage and participation are estimated to produce more post-release defects, meaning that poor code review negatively impacts the software quality. Our study found that code reviews that had found a security defect are associated with a high proportion of comments related to the raised issue (RQ_1), indicating the extra effort and participation that might have been required to find and fix these defects.

Other studies focused on factors that improve the code review quality. For example, Kononenko et al. (2018) empirically examined what factors influence the PR review quality and outcome in the Active Merchant project. They found that the quality of a PR is strongly associated with the quality of its description, its complexity and revertability, while the quality of the review process is linked to the feedback quality, tests quality, and the discussion among developers. Bernardo et al. (2018) examined the impact of adopting CI on the time to integrate PRs. They found that the time to merge PRs increased after adopting CI. In the context of our study, we observed some projects adopting and integrating static analysis tools in the CI pipeline to help in identifying specific security issues (e.g., ReDOS) and other general issues related to fixing the code style and structure, which had not been shown previously by the other aforementioned studies.

Other studies examined how code review is practiced in different contexts (e.g., test code Spadini et al., 2019, 2018, build specifications Nejati et al., 2023). For example, Spadini et al. (2019) examined the impact of a code review practice called 'Test-Driven Code Review' (TDR), where a reviewer inspects patches by examining the changed test code before the changed production code. Their experiments show that developers adopting TDR find more defects than ones found through examining production code. Spadini et al. (2018) also examined how code review is used for ensuring the quality of test code. They find that developers tend to discuss test files significantly less than production files. The paper recommends that the project should set aside sufficient time for reviewing test files. In fact, our study is in-line with such results; we observed that in some cases the test code could help reviewers to identify several security issues (e.g., as seen in this PR Add CounterCache, 2021). Alami et al. (2019) examined the reasons for the success of code review practice, and found that human and social aspects are also key to drive the success. For example, the contributor's passion allows to adhere to code review best practices, and cope with the feedback by learning from rejections and negative comments.

Other most relevant work to our study focuses on security code review (Bacchelli and Bird, 2013; di Biase et al., 2016; Paul et al., 2021; Bosu, 2014). For example, Bacchelli and Bird (2013) observed (based on interviews and surveys) that code review is mainly motivated for finding defects and formatting issues while missing the fact that there were security issues. di Biase et al. (2016) analyzed the Chromium system to understand the factors that may lead to find security issues during code review, and found, for example, that reviews conducted by more than 2 reviewers are being more successful at finding security issues. Also, they found that reviewers tend to find domain-specific security issues (e.g., Cross-Site Scripting XSS) more than language-specific issues (e.g., C++ issues). Our findings show that both of such types (e.g., Race Conditions issues related to C++, overflow issues, and XSS) could be caught during code review of the studied projects.

Several studies focused on analyzing the impact of security vulnerabilities in package ecosystems (Zerouali et al., 2022; Alfadel et al., 2021a, 2022, 2023, 2022). For example, the study by Zerouali et al. (2022) performs a time-based analysis of vulnerabilities reported in the Snyk database. The vulnerabilities in this database are issues found after releasing package versions to the public. This is in steep contrast with the vulnerabilities that we analyze (i.e., found during code review), which can be found and fixed before releasing to the public. We also compare vulnerabilities found during code review to those reported in the advisories dataset (Section 4.2). The aforementioned difference has two main implications. The first implication is that it gives package users an idea of what types of vulnerabilities npm project maintainers are able to identify and fix preemptively. This helps package users better understand the care that is being put by npm project maintainers into the dependencies that these package users are integrating into their own software projects. For example, the data in Table 3, provides an overview of the prevalence

of security-related code reviews in the analyzed systems and motivates research that can bring more awareness about security issues during code reviews. Also, the categories we analyze can serve as a high-level checklist for a security-focused code review. The second implication is that the methodology for finding the vulnerabilities is different. In the Snyk database and npm advisories, there is no clear method that is being used for finding these vulnerabilities. They could be found through bug bounties, through an external contributor who is perusing the code base, or by any other means imaginable. However, this does not give a good idea of how code review is aiding in bettering the security of their packages, if at all.

The study by Paul et al. (2021) analyzed the Chromium OS project. The main goal of the study is to build a regression model on the Chromium project to identify factors that differentiate code reviews with successfully identified vulnerabilities from reviews that missed vulnerabilities. They found, for example, that the number of directories under review correlates negatively with identifying vulnerabilities. Bosu (2014) performed an empirical study, where they analyzed more than 400 vulnerable code changes with the aim to identify their characteristics. They found that the changes by less experienced contributors were significantly more likely to introduce vulnerabilities. Also, they found that new files are less likely to contain vulnerabilities compared to frequently modified files. In our study, we find that the identified security issues are concentrated on a small fraction of the project files (RQ1). This should encourage researchers in the future to understand the nature of such files that frequently contain security issues, which would help practitioners and developers better improve the process of identifying security issues in code review. Recently, Braz and Bacchelli (2022) conducted interviews with 10 professional developers and survey of 182 practitioners to understand (1) what is the current developer's perspective on ensuring software security during code review? (2) to what extent do companies/projects support security assessment during code review? They found that developers do not have security in mind when describing their code review practices, thus suggesting that security is not one of their main priorities when reviewing code. This result partially supports our findings where we find that security issues identified during code review are raised in a small fraction of PRs, affecting also a small fraction of project files. Our study complements the previous works since, we specifically focus on JavaScript projects that have been published in the npm ecosystem, analyzing security issues identified during code review. We also add to previous work by qualitatively studying how developers discuss the raised security issues and tackle them during the review phase (RQ3). Moreover, we present a novel discussion on the comparison of security issues identified during code review to the post-release security issues (advisories). Our study aims to help the community better understand the types of security issues discovered during code review in order to pay attention to them in the future, and understand the mitigation strategies employed by project maintainers to tackle the issues. Our results highlight several observations that aim to increase the awareness of practitioners and researchers to the role of code review in relation to security.

6. Threats to validity

In this section, we discuss the threats to validity of our study. **Internal validity** concerns factors that might affect the casual relationship and experimental bias. In RQ2 and RQ3, we conducted major manual process to extract the required information for analyzing the security issues in the PRs. Like any human activity, our manual process is subject to the author bias. To mitigate this, two authors independently analyze the PRs using an open card-sort method. Moreover, we report a high-level of agreement which indicates that our results are more likely to hold. Additionally, both annotators meet and discuss any conflicts to reach a consensus. Finally, at an early stage of the process, we invite two annotators (who are authors of this paper) to independently investigate a sample of the PRs (i.e., PRs in the first round), and applied their feedback in later rounds. This gives us a high confidence of the data used in our analysis.

Another internal threat to validity is related to our manual process to quantify the number of unique files that contain the identified security issue per PR. We use our best judgment during the manual inspection. For example, if a security issue is raised in a comment on a specific snippet of code, then the file associated with that code snippet would be considered the unique file. Another example would be, if a security issue is raised during the discussion of the design of the changes made in the PR, then all the source code files would be considered. A final example would be if developers are talking about package dependencies, then we would include all files touched that relate to package dependencies. Generally, this is a straightforward process, but in some edge cases, it also requires going a little bit deeper, and hence, we inspect the code to understand the context.

Our keyword-based technique to identify security-related PRs could be another limitation. We may miss security issues in the PRs if the review comments do not contain any of the keywords that we used. That said, our keyword set is curated in an extensive process, by utilizing a well-known set of security-related keywords, which has been used in prior studies (e.g., Paul et al. (2021) and Bosu (2014)). Then, we manually examine the relevance of each keyword and include ones that yield a good relevance (see Section 2.2). Hence, we believe that our keyword set is of high quality, and that the potentially missed security issues will not significantly impact our results.

Another aspect worth pointing out is the size of our final dataset. Our detailed inspection step (outlined in Section 2.3), ended us with 171 PRs with security-related code reviews, despite following a similar approach by Paul et al. (2021), which yielded 374 VCC (i.e., Vulnerability Contributing Commits during code review). As a result, our study yielded a lower amount of data than anticipated. Yet, to avoid any bias from modifying the project filtration process (Section 2.1), we did not seek to add more projects once the initial results were obtained and the final data set was found. The aim is to avoid any further influence from authors in trying to filter for projects that may favor a certain outcome. However, as a case study, we focus our contribution on deriving useful findings from our heavy manual analysis. Therefore, we still believe our findings will be useful in further research and helpful to practitioners who want greater insights into the security issues faced and handled by npm project maintainers.

Finally, in our analysis, we used the PR feature in GitHub to search for security issues raised by project maintainers during code review. However, there might be other security issues that are not discussed through PR feature. That said, through our manual analysis, we did not observe any case where a project maintainer refers to an issue being discussed through other platforms. Therefore, we had to rely on the PR discussion as the main source of information for the security issues raised during code review.

External validity are related to the generalizability of our findings. Our projects dataset contains a limited number of JavaScript projects available in the npm advisories dataset. Hence, it is possible that there are other projects not included in our dataset, which might also be of our interest in this study. However, our projects dataset is of high quality, since we leveraged some

filtration criteria to provide a good representation of the projects we are interested in studying. The projects chosen for our study include popular open-source projects that vary across domains, languages, age, and having high activity level. Also, the number of projects in our dataset is in-line with the similar studies (Paul et al., 2021; Bosu, 2014; Ebert et al., 2019) that also require similar manual process, given the extensive manual analysis required for the study analysis and data collection process, which makes it infeasible to include a lot of projects. Therefore, we believe most of these results can hold for other OSS projects.

We only collected data through open source review discussions, which may not provide the full picture of security perspective during a code review (e.g., developers' perceptions). To mitigate this limitation, one can collect qualitative data from interviews and surveys to understand developers' main challenges to ensure security during code reviews. Although several works has partially covered this area (e.g., Braz and Bacchelli (2022)), we believe that future research can be inspired by our results and triangulate selected findings with other data sources to enhance prior studies in that direction.

Construct Validity. In our study (RQ_1), we aim to capture the prevalence of security issues being raised during code review. To achieve this, we used the metadata of each pull request (PR) to identify the files that contain security issues and the comments that discuss these issues. While we agree that this approach may not fully capture the phenomena we set out to measure, we selected a broad range of measurements (including the number of security-related PRs per project, the number of unique files that contain identified security issues per PR, and the number of comments that specifically discuss security issues per PR) that we believe are meaningfully representative of the underlying phenomena of interest.

Reliability Validity. Threats to reliability validity correspond to the degree to which the same data would lead to the same results when repeated. To the best of our knowledge, our study is the first attempt to quantitatively and qualitatively investigate the role of code review with respect to security issues; hence no ground truth exists to compare our findings in the proposed RQs. We defined the ground truth through the agreement or disagreement of the raters for our investigation.

7. Conclusion

This paper conducts a study to explore the role of code review from the security perspective, by analyzing ten JavaScript open-source GitHub projects.

First, we quantify the prevalence of security issues raised in the project Pull Requests (PRs). Our manual analysis (RQ_1) identified 171 PRs with security-related reviews, which represent a small proportion of all PRs in the studied projects. However, such issues are discussed by project maintainers at length. Between 4.82%–28% of all comments in the 171 PRs are related specifically to the security related concern. Moreover, our manual analysis showed 14 types of security issues raised in code review (RQ_2). In particular, we observe that code review is effective at identifying certain types of security issues, e.g., Race Condition, Access Control, Deadlock and Documentation. When analyzing how project maintainers respond to the raised security issues (RQ_3), we find that the majority of the identified security issues are fixed and mitigated. Yet, the project maintainers sometimes do not fix the issue, due to its technical complexity. Interestingly, sometimes the project maintainers discuss security issues that are not directly related to the reviewed PR. Finally, we present some implications for practitioners and researchers, which aim to support the role code review plays in bettering software security.

We believe that there are several possible directions for future work based on our results. First, our study found some tools that aided in the code review process at helping to find security defects. Our findings may not be conclusive, so future work can seek to further evaluate the effectiveness of these tools and evaluate what types of tools are actually being used in the community to help improve their code reviews for security purposes. Second, documentation security issues are cases we could not find in related work. This is perhaps a result of not directly impacting the security of the project itself, but of the users of the project. Future work should seek to examine how these issues impact the end-user and analyze how the issue comes about more closely. Finally, an obvious direction to improve upon our study would be to gather more resources to collect more data and projects across a wider range of languages and domains. Our work focused on a single ecosystem and language, so a larger breadth of ecosystems might be able to bring greater insights that could not otherwise be seen from the projects we studied.

CRediT authorship contribution statement

Mahmoud Alfadel: Conceptualization, Methodology, Scripting, Data curation, Writing – original draft. **Nicholas Alexandre Nagy:** Conceptualization, Data curation, Investigation. **Diego Elias Costa:** Conceptualization, Supervision, Writing – review & editing. **Rabe Abdalkareem:** Conceptualization, Writing – review & editing. **Emad Shihab:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

The authors have no conflict of interests.

Data availability

The data is available here https://zenodo.org/record/7538187#.Y9IYE-zMKEs.

References

2021. 2834 - Adjust placement of generated (x) button on contextual action panels by EdwardCoyle · pull request #2984 · infor-design/enterprise. https://github.com/infor-design/enterprise/pull/2984, (Accessed on 08/07/2021).

2021. 6935 Remove order string syntax by mkaufmaner · pull request #7160 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/7160, (Accessed on 09/06/2021).

2021. A5:2017-broken access control—OWASP. https://owasp.org/www-project-top-ten/2017/A5_2017-Broken_Access_Control, (Accessed on 09/07/2021).

Abdalkareem, R., 2017. Reasons and drawbacks of using trivial npm packages: The developers' perspective. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, pp. 1062–1064.

2021a. Add connection specific errors by DavidTPate · pull request #2576 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/2576, (Accessed on 06/17/2021).

2021b. Add connection specific errors by DavidTPate · pull request #2576 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/2576, (Accessed on 06/16/2021).

2021. Add: CounterCache feature to hasmany models by kuzmin · pull request #2375 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/2375, (Accessed on 06/17/2021).

2023. Add doc about render props by mjackson · pull request #10741 · facebook/react. https://github.com/facebook/react/pull/10741, (Accessed on 01/25/2023).

2021. Add information on how to possibly fix database connection errors. by matbrgz · pull request #3163 · strapi/strapi. https://github.com/strapi/strapi/pull/3163, (Accessed on 09/05/2021).

2021. Add optional field role if want to register specific user role by MachiAngel · pull request #3201 · strapi/strapi. https://github.com/strapi/strapi/pull/3201, (Accessed on 09/01/2021).

2021. Add rate limit on auth routes by lauriejim · pull request #1681 · strapi/strapi. https://github.com/strapi/strapi/pull/1681, (Accessed on 10/02/2021).
2023. Add reset feature to batch node by HiroyasuNishiyama · pull request #2553 · node-red/node-red. https://github.com/node-red/node-red/pull/2553, (Accessed on 01/25/2023).
2021. Add snyk badge by styfle · pull request #1420 · markedjs/marked. https://github.com/markedjs/marked/pull/1420, (Accessed on 10/09/2021).
2021. added data: link fix to prevent xss by matt- · pull request #844 · markedjs/marked. https://github.com/markedjs/marked/pull/844, (Accessed on 10/09/2021).
2023. Adding support for populating db outputted values inserted with sequelize.literal by renatoargh · pull request #6871 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/6871, (Accessed on 01/25/2023).
Alami, A., Cohn, M.L., Wasowski, A., 2019. Why does code review work for open source software communities? In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 1073–1083.
Alfadel, M., 2023. Qualitative analysis of security-related code reviews in npm packages: An empirical study—zenodo. https://zenodo.org/record/7538187#.Y8OoVuzMKEs, (Accessed on 01/15/2023).
Alfadel, M., Costa, D.E., Mokhallalati, M., Shihab, E., Adams, B., 2020. On the threat of npm vulnerable dependencies in node. js applications. arXiv preprint arXiv:2009.09019.
Alfadel, M., Costa, D.E., Shihab, E., 2021a. Empirical analysis of security vulnerabilities in python packages. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 446–457.
Alfadel, M., Costa, D.E., Shihab, E., 2023. Empirical analysis of security vulnerabilities in python packages. Empir. Softw. Eng. 28 (3), 59.
Alfadel, M., Costa, D.E., Shihab, E., Adams, B., 2022. On the discoverability of npm vulnerabilities in node. js projects. ACM Trans. Softw. Eng. Methodol..
Alfadel, M., Costa, D.E., Shihab, E., Mkhallalati, M., 2021b. On the use of dependabot security pull requests. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 254–265.
2023. Allow read-access to protectedfields based on user for custom classes by dobbias · pull request #5887 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/5887, (Accessed on 01/25/2023).
Aloraini, B., Nagappan, M., German, D.M., Hayashi, S., Higo, Y., 2019. An empirical study of security warnings from static application security testing tools. J. Syst. Softw. 158, 110427.
Bacchelli, A., Bird, C., 2013. Expectations, outcomes, and challenges of modern code review. In: 2013 35th International Conference on Software Engineering. ICSE, IEEE, pp. 712–721.
Beller, M., Bacchelli, A., Zaidman, A., Juergens, E., 2014. Modern code reviews in open-source projects: Which problems do they fix? In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 202–211.
Bernardo, J.H., da Costa, D.A., Kulesza, U., 2018. Studying the impact of adopting continuous integration on the delivery time of pull requests. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories. MSR, IEEE, pp. 131–141.
2021. Block non-'file://' URLs when 'nodeintegration' is enabled by poiru · pull request #9224 · electron/electron. https://github.com/electron/electron/pull/9224, (Accessed on 08/13/2021).
Bosu, A., 2014. Characteristics of the vulnerable code changes identified through peer code review. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 736–738.
Bosu, A., Carver, J.C., 2013. Peer code review to prevent security vulnerabilities: An empirical evaluation. In: 2013 IEEE Seventh International Conference on Software Security and Reliability Companion. IEEE, pp. 229–230.
Bosu, A., Carver, J.C., Hafiz, M., Hilley, P., Janni, D., 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 257–268.
Braz, L., Bacchelli, A., 2022. Software security during modern code review: the developer's perspective. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 810–821.
2021. Changelog for 16.9 by gaearon · pull request #16254 · facebook/react. https://github.com/facebook/react/pull/16254, (Accessed on 08/13/2021).
2023. Changelog for 16.9 by gaearon · pull request #16254 · facebook/react. https://github.com/facebook/react/pull/16254, (Accessed on 01/25/2023).
Cohen, J., 1960. A coefficient of agreement for nominal scales. Educ. Psychol. Meas. 20 (1), 37–46.
2021. CWE - common weakness enumeration. https://cwe.mitre.org/index.html, (Accessed on 07/10/2021).
Davis, J.C., Coghlan, C.A., Servant, F., Lee, D., 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 246–256.
2021. davisjam/vuln-regex-detector: Detect vulnerable regexes in your project. REDOS, catastrophic backtracking. https://github.com/davisjam/vuln-regex-detector#readme, (Accessed on 08/12/2021).
Dey, T., Mockus, A., 2020. Effect of technical and social factors on pull request quality for the npm ecosystem. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–11.
di Biase, M., Bruntink, M., Bacchelli, A., 2016. A security perspective on code review: The case of chromium. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation. SCAM, IEEE, pp. 21–30.
2021. Docs: Update application-architecture.md by AqibMukhtar · pull request #23650 · electron/electron. https://github.com/electron/electron/pull/23650, (Accessed on 08/07/2021).
2023. docs: Update application-architecture.md by AqibMukhtar · pull request #23650 · electron/electron. https://github.com/electron/electron/pull/23650, (Accessed on 01/25/2023).
2021a. Don't quote order columns that are functions by seth-admittedly · pull request #783 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/783, (Accessed on 09/06/2021).
2021b. Don't quote order columns that are functions by seth-admittedly · pull request #783 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/783, (Accessed on 06/17/2021).
Ebert, F., Castor, F., Novielli, N., Serebrenik, A., 2019. Confusion in code reviews: Reasons, impacts, and coping strategies. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 49–60.
2021. electron - npm. https://www.npmjs.com/package/electron, (Accessed on 09/12/2021).
2021. enable CommonMark spec 468 by trott · pull request #1305 · markedjs/marked. https://github.com/markedjs/marked/pull/1305, (Accessed on 09/07/2021).
Equifax, 2021. Equifax releases details on cybersecurity incident, announces personnel changes—equifax. URL https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832, Accessed on 01/12/2021.
2021. Escape component keys used in reactid by syranide · pull request #714 · facebook/react. https://github.com/facebook/react/pull/714, (Accessed on 08/12/2021).
2021. escapeTextContentForBrowser no longer escapes 'and", quoteAttributeValueForBrowser no longer escapes ' by syranide · pull request #3152 · facebook/react. https://github.com/facebook/react/pull/3152, (Accessed on 09/01/2021).
2023. feat: add a new contextbridge module by MarshallOfSound · pull request #20307 · electron/electron. https://github.com/electron/electron/pull/20307, (Accessed on 01/25/2023).
2021. [Feature] add file upload policy by jaeggerr · pull request #4822 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/4822, (Accessed on 06/16/2021).
Fincher, S., Tenenberg, J., 2005. Making sense of card sorting data. Expert Syst. 22 (3), 89–93.
2023. fix: make run-gn-format work properly on windows by brenca · pull request #18993 · electron/electron. https://github.com/electron/electron/pull/18993, (Accessed on 01/25/2023).
2021. Fix deep me data graphql query by lauriejim · pull request #4790 · strapi/strapi. https://github.com/strapi/strapi/pull/4790, (Accessed on 09/01/2021).
2021. Fix deep me data graphql query by lauriejim · pull request #4790 · strapi/strapi. https://github.com/strapi/strapi/pull/4790, (Accessed on 09/05/2021).
2021a. Fix GFM tables not breaking on block-level structures by calculuschild · pull request #1598 · markedjs/marked. https://github.com/markedjs/marked/pull/1598, (Accessed on 09/07/2021).
2021b. Fix GFM tables not breaking on block-level structures by calculuschild · pull request #1598 · markedjs/marked. https://github.com/markedjs/marked/pull/1598, (Accessed on 06/06/2021).
2021a. fix image links with escaped brackets by UziTech · pull request #1683 · markedjs/marked. https://github.com/markedjs/marked/pull/1683, (Accessed on 09/06/2021).
2021b. Fix image links with escaped brackets by UziTech · pull request #1683 · markedjs/marked. https://github.com/markedjs/marked/pull/1683, (Accessed on 09/07/2021).
2023. Fix ReDOS #1405 by feder1co5oave · pull request #1408 · markedjs/marked. https://github.com/markedjs/marked/pull/1408, (Accessed on 01/25/2023).
2023. fix(postgres): check and enable standard conforming strings when required by sushantdhiman · pull request #10746 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/10746, (Accessed on 01/25/2023).
2021. fix(redshift): allow standard_conforming_strings option by aheuermann · pull request #10816 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/10816, (Accessed on 06/17/2021).

Fleiss, J.L., Cohen, J., 1973. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. Educ. Psychol. Meas. 33 (3), 613–619.
2023. GitHub apps - debricked. https://github.com/apps/debricked, (Accessed on 01/23/2023).
2023. GitHub integration - snyk user docs. https://docs.snyk.io/integrations/git-repository-scm-integrations/github-integration, (Accessed on 01/23/2023).
2021. github/codeql: CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub advanced security (code scanning), LGTM.com, and LGTM enterprise. https://github.com/github/codeql, (Accessed on 10/06/2021).
2021. Improve ctm edit by soupette · pull request #685 · strapi/strapi. https://github.com/strapi/strapi/pull/685, (Accessed on 08/07/2021).
2021a. Improve soundness of ReactDOMFiberInput typings by philipp-spiess · pull request #13367 · facebook/react. https://github.com/facebook/react/pull/13367, (Accessed on 09/01/2021).
2021b. Improve soundness of ReactDOMFiberInput typings by philipp-spiess · pull request #13367 · facebook/react. https://github.com/facebook/react/pull/13367#issuecomment-412349795, (Accessed on 09/01/2021).
Imtiaz, N., Thorne, S., Williams, L., 2021. A comparative study of vulnerability reporting by software composition analysis tools. arXiv preprint arXiv:2108.12078.
2021. Infor-design/enterprise-ng: Angular wrappers for IDS enterprise components. https://github.com/infor-design/enterprise-ng, (Accessed on 09/12/2021).
2023. IQ server. https://help.sonatype.com/iqserver, (Accessed on 01/23/2023).
Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2014. The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101.
Kononenko, O., Rose, T., Baysal, O., Godfrey, M., Theisen, D., De Water, B., 2018. Studying pull request merges: a case study of shopify's active merchant. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. pp. 124–133.
Li, S., Kang, M., Hou, J., Cao, Y., 2021. Detecting node. js prototype pollution vulnerabilities via object lookup analysis. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 268–279.
Mäntylä, M.V., Lassenius, C., 2008. What types of defects are really discovered in code reviews? IEEE Trans. Softw. Eng. 35 (3), 430–448.
2021. Marked - npm. https://www.npmjs.com/package/marked, (Accessed on 09/12/2021).
McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 192–201.
McIntosh, S., Kamei, Y., Adams, B., Hassan, A.E., 2016. An empirical study of the impact of modern code review practices on software quality. Empir. Softw. Eng. 21 (5), 2146–2189.
2021. Merge pull request from GHSA-4w46-w44m-3jq3 · parse-community/parse-server@da905a3. https://github.com/parse-community/parse-server/commit/da905a357d062ab4fea727a21eac231acc2ed92a, (Accessed on 09/20/2021).
2021. Messaging API support of core nodes by k-toumura · pull request #2402 · node-red/node-red. https://github.com/node-red/node-red/pull/2402, (Accessed on 06/17/2021).
2021. Migrate ReactSuspense fuzz tests to property based tests by dubzzz · pull request #18673 · facebook/react. https://github.com/facebook/react/pull/18673, (Accessed on 06/17/2021).
Mirhosseini, S., Parnin, C., 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 84–94.
2021. moment - npm. https://www.npmjs.com/package/moment, (Accessed on 09/12/2021).
Nejati, M., Alfadel, M., McIntosh, S., 2023. Code Review of Build System Specifications: Prevalence, Purposes, Patterns, and Perceptions. In: Proc. of the International Conference on Software Engineering. ICSE.
2021. nested parentheses link by UziTech · pull request #1414 · markedjs/marked. https://github.com/markedjs/marked/pull/1414, (Accessed on 10/09/2021).
2021. node-red - npm. https://www.npmjs.com/package/node-red, (Accessed on 09/12/2021).
2023. nodesecurity/nsp: node security platform command-line tool. https://github.com/nodesecurity/nsp, (Accessed on 01/23/2023).
2020. npm. https://www.npmjs.com/advisories, (Accessed on 11/02/2020).
2021. npm. https://www.npmjs.com/advisories, (Accessed on 09/08/2021).
2020. npm - libraries.io. https://libraries.io/npm, (Accessed on 11/02/2020).
2021. parse-server - npm. https://www.npmjs.com/package/parse-server, (Accessed on 09/12/2021).
Paul, R., Turzo, A.K., Bosu, A., 2021. Why security defects go unnoticed during code reviews? a case-control study of the chromium os project. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 1373–1385.
2021. Protected fields pointer-permissions support by dobbias · pull request #5951 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/5951, (Accessed on 06/06/2021).
2023. protocol: cleanup by deepak1556 · pull request #2125 · electron/electron. https://github.com/electron/electron/pull/2125, (Accessed on 01/25/2023).
2021. react - npm. https://www.npmjs.com/package/react, (Accessed on 09/12/2021).
2021. Reload grant auth config into db when add/delete grant provider by hanyuei · pull request #991 · strapi/strapi. https://github.com/strapi/strapi/pull/991, (Accessed on 08/07/2021).
2021. Render html in heading by UziTech · pull request #1622 · markedjs/marked. https://github.com/markedjs/marked/pull/1622, (Accessed on 05/30/2021).
Rennhard, M., Kushnir, M., Favre, O., Esposito, D., Zahnd, V., 2022. Automating the detection of access control vulnerabilities in web applications. SN Comput. Sci. 3 (5), 376.
2021. Robust animation-end handling in ReactCSSTransitionGroup by djrodgerspryor · pull request #4561 · facebook/react. https://github.com/facebook/react/pull/4561, (Accessed on 08/07/2021).
2021. Roles should follow acls by georgesjamous · pull request #4895 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/4895, (Accessed on 06/16/2021).
2021a. security: fix REDOS vulnerabilities by davisjam · pull request #1083 · markedjs/marked. https://github.com/markedjs/marked/pull/1083#issuecomment-368726539, (Accessed on 09/07/2021).
2021b. security: fix unsafe heading regex by davisjam · pull request #1224 · markedjs/marked. https://github.com/markedjs/marked/pull/1224, (Accessed on 08/08/2021).
2021. sequelize - npm. https://www.npmjs.com/package/sequelize, (Accessed on 09/12/2021).
Software, B.D., 2019. Synopsys black duck open source security and risk analysis.
Spadini, D., Aniche, M., Storey, M.-A., Bruntink, M., Bacchelli, A., 2018. When testing meets code review: Why and how developers review tests. In: 2018 IEEE/ACM 40th International Conference on Software Engineering. ICSE, IEEE, pp. 677–687.
Spadini, D., Palomba, F., Baum, T., Hanenberg, S., Bruntink, M., Bacchelli, A., 2019. Test-driven code review: an empirical study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering. ICSE, IEEE, pp. 1061–1072.
2021. Stack overflow developer survey 2019. https://insights.stackoverflow.com/survey/2019, (Accessed on 08/26/2021).
2021. strapi - npm. https://www.npmjs.com/package/strapi, (Accessed on 09/12/2021).
2021a. Structured /health response by montymxb · pull request #4305 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/4305, (Accessed on 09/05/2021).
2021b. Structured /health response by montymxb · pull request #4305 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/4305/commits/be4aef061d7b1967ce020e286304e08a6ac389d2, (Accessed on 09/05/2021).
2023. Structured /health response by montymxb · pull request #4305 · parse-community/parse-server. https://github.com/parse-community/parse-server/pull/4305, (Accessed on 01/25/2023).
2021. test: security scan by davisjam · pull request #1220 · markedjs/marked. https://github.com/markedjs/marked/pull/1220, (Accessed on 09/07/2021).
Thongtanunam, P., McIntosh, S., Hassan, A.E., Iida, H., 2015. Investigating code review practices in defective files: An empirical study of the qt system. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 168–179.
2021. uglify-js - npm. https://www.npmjs.com/package/uglify-js, (Accessed on 09/12/2021).
2021. Unable to override role while creating user through API o…by skavinvarnan · pull request #5330 · strapi/strapi. https://github.com/strapi/strapi/pull/5330, (Accessed on 09/01/2021).
2021. Update to resolve js-yaml vulnerability by calvinchengx · pull request #1472 · markedjs/marked. https://github.com/markedjs/marked/pull/1472, (Accessed on 08/08/2021).
2023. Update travis-CI to use docker by toddbluhm · pull request #6443 · sequelize/sequelize. https://github.com/sequelize/sequelize/pull/6443, (Accessed on 01/25/2023).
Walden, J., 2020. The impact of a major security event on an open source project: The case of OpenSSL. In: Proceedings of the 17th International Conference on Mining Software Repositories. pp. 409–419.
2021. What is threat hunting?—IBM. https://www.ibm.com/topics/threat-hunting, (Accessed on 11/03/2021).

Yang, J., Tan, L., Peyton, J., Duer, K.A., 2019. Towards better utilizing static application security testing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 51–60.

Zerouali, A., Mens, T., Decan, A., De Roover, C., 2021. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. arXiv preprint arXiv:2106.06747.

Zerouali, A., Mens, T., Decan, A., De Roover, C., 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. Empir. Softw. Eng. 27 (5), 1–45.

Zimmermann, M., Staicu, C.-A., Tenny, C., Pradel, M., 2019. Small world with high risks: A study of security threats in the npm ecosystem. In: 28th {USENIX} Security Symposium ({USENIX} Security 19). pp. 995–1010.

Mahmoud Alfadel is a postdoctoral researcher at the REBELs Lab, in the Cheriton School of Computer Science at the University of Waterloo. His research interests cover a wide range of software engineering-related topics, including mining software repositories, empirical software engineering, software ecosystems, and release engineering.

Nicholas Alexandre Nagy is a DevOps Engineer at EXFO. Before that he joined the DAS Lab as a research assistant under the NSERC Undergraduate Research Student Award. His main research interests relate to Software Quality Assurance, Mining Software Repositories and Machine Learning.

Diego Elias Costa is an assistant professor in the Computer Science department at UQAM in Montréal, Canada. He is also part of the LATECE research group. Before that, he was a postdoctoral researcher at Concordia University, Canada, working with Prof. Emad Shihab. He received his Ph.D. from Heidelberg University, Germany in the Parallel and Distributed Systems Group of Prof. Andrzejak. He is interested in Software Engineering Research. In a few words, he wants to reduce the burden on software developers by tackling aspects related to software maintenance and software performance.

Rabe Abdalkareem received his Ph.D. in Computer Science and Software Engineering from Concordia University. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. Abdalkareem received his master's in applied computer science from Concordia University. His work has been published at premier venues such as FSE, ICSME, and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. You can find more about him at http://users.encs.concordia.ca/rababdu.

Emad Shihab is a professor in the Department of Computer Science and Software Engineering at Concordia University. He received his Ph.D. from Queens University. Dr. Shihabs research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.