Method-Level Data in GitHub Pull Request Descriptions: Effects on Developers' Prioritization and Facilitation of Fixing Vulnerable Dependencies

Tudor-Alexandru Popovici , Mehdi Keshani , Sebastian Proksch
TU Delft

Abstract

Modern software development involves the usage of external third-party software projects as direct dependencies. Nonetheless, developers of a dependant project have no control over critical aspects such as development and testing of the dependency. This can put the reliant repositories at risk through vulnerabilities, which can be exploited by malicious attackers. Automated dependency maintenance tools can mitigate the risks, but have an observed shortcoming: they have decreased vulnerability detection accuracies due to their package-level analysis approach.

In this study, a total of 6.717 active projects hosted on GitHub have been analysed using a method-level vulnerability analysis, discovering 24 projects affected by 4 distinct exposures. The developers have been notified through GitHub Pull Requests, which contained the methods in their projects that called vulnerable dependency methods. This was done with the aim of finding answers to: (i) whether the provided method call information makes developers prioritize the task of fixing vulnerabilities, (ii) whether the fine-grained information facilitates the exposures handling process. Developers' reactions to the method-level data were collected through means of a survey. Collected data revealed that the fine-grained information in the PRs did have a positive effect on the developers' prioritization of fixing the vulnerable dependencies. Moreover, the provided data also facilitated the maintainers' fix process to some extent. However, due to the limited amount of recorded responses, the answer to the research question could not be concluded.

Keywords— Dependant, Dependency, Vulnerability, Dependency maintenance tools, Package-level, Method-level, GitHub Pull Requests

1 Introduction

Most, if not all contemporary software projects make use of third-party libraries, known as dependencies. Dependencies can speed up the software development process, as all of the potential developer's workload in writing the code provided by the dependency is outsourced to the developers of that project [1].

Dependencies themselves are software projects and are thus inherently prone to having bugs. In fact, it is reported that half of the total time a developer allocates working on a project is spent on fixing bugs [2]. Vulnerabilities, also known as bugs, represent "a big threat for the security of software systems" [3], mainly due to the fact that malicious attackers can exploit these exposures for their own benefit.

In order to mitigate the risks vulnerable dependencies can impose on dependant projects, exposed packages must be updated to versions which had their bugs fixed. Failure to update vulnerable dependencies in time can result in serious consequences on the projects. For example, more than 100.000 credit card user details were leaked by company Equifax [4], due to the developers' inability of promptly updating the vulnerable Apache Struts package. Such incidents can actively be prevented by periodically practicing the process of dependency maintenance.

Dependency maintenance, the process of identifying and updating dependencies that are vulnerable or outdated has nowadays been automatised. An example of a popular dependency maintenance tool is Dependabot [5], which is available for use for any project hosted on GitHub. Although the uncontested benefits of Dependabot, it has been observed by numerous developers that this type of maintenance tool generates a high amount of false positives (FPs) and low severity alerts [6]. The developers have also suggested that these analysis tools could benefit from having this aspect improved.

The percentage of FP results (projects which are marked as having vulnerable dependencies) has to do with the type of analysis Dependabot is performing. Vulnerable dependencies are searched in a project on the package-level. Nonetheless, the actual vulnerable methods inside of the exposed packages might not even be called from the reliant repositories, leading to a FP result.

Several studies have shown the possibility of lowering the amount of FPs through the use of call graph (CG) data structures [7][4]. The use of these data structures enables the mapping of method calls between the dependant project and vulnerable dependency methods. As a result, a more fine-grained type of analysis, known as method-level analysis can be performed on projects.

The main interest of the study is how the developers of a

repository that is affected by a vulnerability react to method-level information (the set of methods from the dependant repository which call vulnerable methods in the dependency) provided in GitHub Pull Request (PR) descriptions. Specifically, the following research question (RQ) will be answered:

What are the effects of method-level vulnerability information, given in GitHub Pull Request descriptions, on the prioritization and facilitation of the process of fixing the vulnerable dependencies by the developers, in the scope of their affected projects?

Primarily, it is of interest to see whether this type of vulnerability information has any effect on how developers prioritize fixing the dependencies (when compared to past exposure fixing behaviours). Moreover, it is relevant whether the knowledge of which methods are at risk facilitates the developers' process of fixing exposures. To the best of the writer's knowledge, no previous research has been conducted on the effects of fine-grained vulnerability data on developers' of affected projects.

The structure of this research paper will be as follows. Section 2 will discuss the background of the methods that have been used in this study. Section 3 will introduce the methodology that was followed in order to answer the research question presented in the introduction. The 4th Section will guide the reader through the experimental setup of the research (how the methodology has been put into practice). Moreover, it will provide the results of the study, which will give the answer to the RQ. Section 5 will provide insights on the ethical aspects of the research. In addition, it will illustrate the extent of reproducibility of the research that has been carried out. This section will be followed by a discussion of the results, which will reflect and compare the results to other relevant research studies. Lastly, Section 7 will provide the conclusion and will discuss possible future work emerging from this study.

2 Research Background

Vulnerabilities are defined as "weaknesses or flaws present in your code" [8]. These weaknesses are caused by incorrectly written source code, causing program logic gaps that lead to security exposures that can be exploited by malicious attackers.

With the aim of minimizing the risks these exposures impose on projects and their users, a collaborative process between vulnerability researchers and project maintainers, called vulnerability disclosure, is nowadays commonly practiced [9]. The main workflow of this process starts with the security researchers that notify the maintainers of the found vulnerabilities affecting their projects. This is followed by the developers creating a corresponding patch and then notifying all of the other repositories which might depend on their project.

Vulnerabilities are publicly disclosed through the use of security advisories. Examples of such advisories are the National Vulnerability Database [10], GitHub Security Advisory [11] or the npm Security Advisory [12].

These are databases that include an extensive number of vulnerability records, that are displayed to the developers in a standardized Common Vulnerabilities and Exposures (CVE) [13] format. These CVEs allow security reporters, repository maintainers and other developers to communicate more efficiently about a specific vulnerability. This is due to the fact that each of these CVEs is assigned to a unique identification number (with format *CVE-{YEAR}-{ID}*, i.e. CVE-2019-1234), that allows easy referral. Moreover, this format includes information such as the version range of the linked package that is affected, the severity level of the exposure and a description of the actual vulnerability. The severity is calculated using the Common Vulnerability Scoring System (CVSS), which produces a base score that is mapped to one of the 4 possible severity levels: *LOW, MEDIUM, HIGH, CRITICAL* [14].

Dependency management tools make extensive use of these security advisories in order to keep projects secure. Dependabot is a popular example of such a tool, having over 7.5 million merged GitHub Pull Requests associated to vulnerability fixes [5]. This tool aims to keep a project's dependencies up-to-date and vulnerability-free. To keep a project's dependencies secure, Dependabot pulls the dependency files of that repository and checks whether any listed dependencies matches any known vulnerabilities from GitHub's Security Advisory (dependency versions need to match vulnerability affected ranges). If a match is found, Dependabot will open a PR that bumps the dependency to the closest version that has the vulnerability patched [15].

Dependabot's strategy of finding vulnerabilities in a project follows a package-level approach. This approach only analyses whether vulnerable versions are in the dependency graph of a project, but does not check whether any methods that cause the vulnerability in the dependency package are actually called from the project. This can lead to situations where a project is marked as having a vulnerable dependency, when none of the vulnerable methods are called, producing a FP result.

A viable solution to the problem of FPs would be performing method-level analysis on the projects. This analysis extends on the package-level one, by adding an extra check of whether the vulnerable methods in the affected dependency are called from the inspected repository. This approach requires 2 main components in order to be carried out:

- There needs to be a way through which it can be determined which methods have been chain-called from a project.
- A data source containing the set of methods linked to specific vulnerabilities needs to be used.

A pragmatic choice for the first component is represented by the CG data structures. These are directed graphs that map function calls both internally within a project and externally between multiple projects.

Furthermore, a data source at the disposal of this study is the Fine-Grained Analysis of Software Ecosystems as Networks (FASTEN) Project's database [16], which includes a vast amount of package-level and method-level vulnerability data (including methods linked to exposures) collected on nu-

merous packages. These projects are all packages released on the Maven Central Repository [17], which is a remote host of Open Source Software (OSS) Java based libraries.

3 Methodology

This section will elaborate on the methodology that was used in order to answer the research question. The goal is to collect data on how developers react to vulnerable method-level information affecting their projects. With this aim in mind, the scope of the study needs to be narrowed down to a finite set of projects on which both package and method level analyses can be performed. With the results from the method-level analysis, GitHub PRs will be opened for the affected repositories. Surveys will then be sent to the developers in order to gather data on how their vulnerability fix process experience was influenced by the fine-grained data. In a more systematic manner, the following steps describe the employed methodology:

1. Select the set of projects on which to perform the study.
2. Retrieve a set of vulnerable packages to look for as dependencies in the selected projects.
3. Implement a vulnerability analyzer to perform package and method-level analyses on selected projects.
4. Analyse the selected repositories for vulnerabilities on the package-level.
5. Analyse the positive package-level repositories for vulnerabilities on the method-level.
6. Open PRs on GitHub for projects which are vulnerable on the method-level.
7. Collect and process data on the reactions of the developers of the notified repositories through means of a survey.

3.1 Project Selection

The first step in being able to answer the research question is to find the set of projects on which the study can be conducted. With the use of the FASTEN database, these projects can be extracted. In order to fit the needs of the study, multiple filters are going to be applied on these packages:

- Projects have to be hosted on the GitHub platform.
- Projects have to have recent development activity.
- Projects have to be non-forked.

The first and primary filter that will be applied on the projects is that they are hosted on the GitHub platform. GitHub has been selected as a platform of choice because it provides the Dependabot maintenance tool, which, nowadays is widely used by the projects hosted on this remote repository host. As a result, projects hosted on this platform are more likely to already use such dependency maintenance tools and thus their developers have a higher likelihood of being familiarized to the automatized process of dependency maintenance.

Another filter to apply on the projects, that would help in maximizing the number of responses that are recorded from the developers is the activity of the repositories. Developers of recently active projects have an increased probability to respond to the vulnerability related opened PRs, as projects showing activity well in the past could be linked to archived or simply unmaintained projects. A good measure of recency is the last update date of that repository, which has been chosen not be more than 4 months old in order for the project to be labeled as active.

3.2 Vulnerability Information Retrieval

Complementary to the set of projects, the vulnerable packages to look for in the dependencies of the selected repositories need to be gathered. Through the use of the FASTEN database, it is also possible to extract this data. This data source contains packages that are themselves flagged as vulnerable, along with the exposure information associated with them. Essentially, this information includes the linked CVE to the vulnerability and its associated data and the set of methods contributing to the vulnerability. This enables the possibility of performing package and method-level analyses on the projects.

3.3 Package-level Analysis

An analyzer was implemented to perform package-level inspection. This vulnerability finder performs the analysis on the projects in a similar manner dependency management tools such as Dependabot carry out their analysis of a repository: versioned dependencies are extracted from a project's dependency file and matched with known vulnerabilities.

The implemented inspector supports projects using any of the 2 most popular build automation tools for Java based projects: Maven [18] or Gradle [19]. Maven provides the Project Object Model (POM) [20], which is an XML file with name *pom.xml* that contains general information about the project's structure, including its list of dependencies. In a similar manner, Gradle has an analogous representation of a project's composition through a file named *build.gradle*. Two custom parsers have been implemented that extract the versioned dependencies listed in the dependency files provided by these build automation tools.

When trying to determine whether a given dependency is vulnerable, the inspector will try to match it to any CVE that is linked to it. Afterwards, the given dependency version will be checked whether it is included in the affected version range from the CVE. If it is included, the repository is marked as package-level vulnerable.

3.4 Method-level Analysis

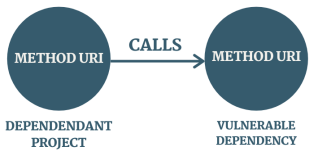

Figure 1: CG call between a project and a vulnerable dependency

Method-level analysis is performed on the projects that were deemed vulnerable by the package-level inspection. Realiza-

tion of this approach is enabled through the construction of call graphs. Specifically, call graphs are generated between the dependant project and each of the package-level vulnerable dependencies.

The call graph will contain edges between two nodes that represent methods, directed from the calling to the called function respectively. Each node contains a Uniform Resource Identifier (URI) that precisely specifies the intra-project location of the method represented by the node. As an example, Figure 1 illustrates an edge between a method in a project that calls a vulnerable method in one of its dependencies.

Given a dependency which is marked as vulnerable by the package-level analysis, the inspector will retrieve all of the methods in the dependency which have caused the CVE to be issued. Starting from these vulnerable methods, the graph will be traversed in the direction of the dependant project, looking for any nodes containing methods of this package. If such a method is found, the repository is concluded to be method-level vulnerable.

3.5 Opening Pull Requests

The method-level analysis will result in a number of different graph path traces that start from the vulnerable methods in the dependency and end at a method in the dependant project. These path traces are part of what is referred to as method-level or fine-grained information.

With all of the generated fine-grained data, the developers of the affected projects need to be notified of the vulnerabilities that put their projects at risk. GitHub PRs will be used, which will bump the version of the vulnerable dependency to a minimum version which has the exposure fixed, mimicking the behaviour of Dependabot. The description of the PR will include all of the method-level vulnerability information collected, including CG traces, linked vulnerability CVE, severity level and release note links.

3.6 Collecting and Processing Developer Responses

The reactions of the developers will be qualitatively analyzed in order to bring answers to the research question. In relation to the effects of method-level data on the prioritization of fixing the vulnerable dependencies, the merging times of the opened PRs will be recorded and compared against the past merging times of other dependency related PRs.

Moreover, developers will be asked to complete a survey, containing both binary (Yes/No) and 5 point Likert Scale [21] answer statements. These statements will cover the degree to which the provided fine-grained data have contributed to the facilitation and prioritization of their vulnerability fix process.

4 Experimental Setup and Results

In this section, the details of the methodology steps laid out in Section 3 will be described. This includes particularities of the selection of the project and vulnerable package sets, analyzer implementation, analysis results retrieval and verification and PR opening process respectively. Moreover, the results of the experimental study will be presented.

4.1 Project Selection Query

For extracting the projects and vulnerable packages, 3 tables provided by the FASTEN database schema [22] have been used:

- **packages**: provides elementary project information, such as the repository Uniform Resource Locator (URL) at which the project is hosted and project name.
- **package_versions**: provides information about the package versions, along with a *metadata* JavaScript Object Notation (JSON) field that contains a *vulnerabilities* property. This latter field provides details about the vulnerabilities that are linked to a package version.
- **dependencies**: provides information on which package depends on what other package, in the form of dependant and dependency.

In order to retrieve the vulnerable projects, a query has been executed, that matched on the dependencies listed in the *dependencies* table having any vulnerable version. Even though there is no explicit field indicating whether a version is vulnerable, this has been checked by verifying that the *metadata* field in the *package_versions* table had its *vulnerabilities* field non-null.

The dependant repositories of these dependencies were then selected, with the query returning a total of 7.638 distinct projects which had at least one vulnerability. Further analysis on the selected projects revealed that 6.717 of these packages had a repository URL that was linking to GitHub.

4.2 Vulnerability Information Retrieval

The Vulnerability Analyzer [23], part of FASTEN Project, gathers vulnerability information from a multitude of security advisory databases. Each retrieved exposure is periodically checked for having any patches being released. When patch commits are detected, the analyzer processes these and determines which methods have been changed. It is then assumed that the changed methods are the ones that have caused the vulnerability.

After the vulnerability is retrieved and a patch for it is found, the data collected is summarized in a Vulnerability Object Definition (VOD) [24] format. This format provides numerous fields, but useful to this study are fields such as linked CVE identifier, vulnerability severity and 2 lists respectively:

- Vulnerable package URLs (purls) [25], which represent a standardized format aimed at reliably locating packages within their package management system.
- Vulnerable method FASTEN URIs, which provide the project path to the functions, including the file containing the method, the method definition, the parameter types and return type respectively.

For each package version linked to a vulnerability, the collected VOD data is then included in the *metadata* field of that vulnerable version in the *package_versions* table.

Using a similar query as the one for extracting the set of projects having vulnerable dependencies, the packages for

which a CVE is issued are retrieved. A total of 211 packages, summing to an overall of 435 CVEs have been found, out of which 393 were unique.

4.3 Analyzer Implementation

An analyzer that checks whether a given project is package and method-level vulnerable has been implemented. It is important that the development version of a project is inspected, as that is typically the most up-to-date version of the repository. Figure 2 illustrates a high level overview of the functionality of the analyzer, splitting it into 7 different main components.

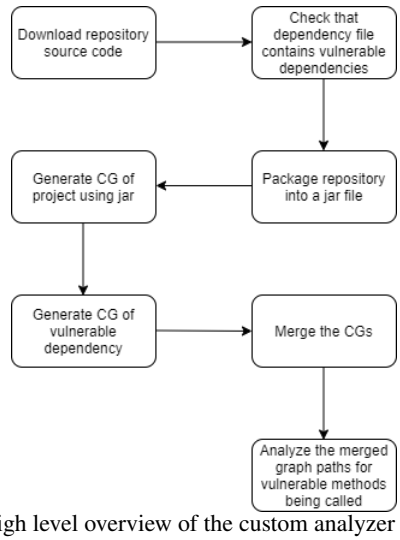

Figure 2: High level overview of the custom analyzer functionality

Scraping GitHub's API and Downloading Repository

A module of the analyzer has been developed as a means of scraping general information from GitHub's API for the selected projects. The data obtained from the API included information such as:

- Default branch name
- Repository user owner
- Repository name
- Total number of stars
- Date at which the repository has last been updated
- Boolean indicating whether the repository is a forked version of another repository

Using the default branch name, the analyzer proceeds by downloading from GitHub a zip file containing the source code of the repository's main branch (which is assumed to correspond to the most up-to-date version of the project). This file is then unzipped and added to a dedicated folder of downloaded repositories.

Dependency File Vulnerability Check

The analyzer's next step is represented by checking whether the dependency file of the project under investigation contains any package that is vulnerable. This step can be seen as a package-level analysis implementation, as only the dependency files are inspected for vulnerabilities.

With the aim of improving performance, the need of downloading a whole repository for inspecting a single dependency file has been eliminated. This has been accomplished by individually downloading the dependency files of the vulnerable projects. By modifying the Dependabot Demonstration Script [26] to return all of the dependency files reachable from the root project directory, both parent and subproject files were retrieved. A total of 17.142 *pom.xml* and 2.855 *build.gradle* dependency files have been downloaded. Furthermore, information such as the locally downloaded dependency file path, the relative path of the dependency file from the root project directory, the dependency file type (Maven or Gradle) and the project in which the dependency file is located have been stored in a file in a *.csv* format.

Using the locally downloaded dependency files, the analyzer can check whether the files contain any vulnerabilities. Depending on the build management tool used by the project (Maven or Gradle), different dependency file parsers have been used. The inspector makes use of the parser to retrieve the list of versioned dependencies, which are then matched with the vulnerable packages extracted from the database.

Packaging the Repository

Packaging the inspected repository in a Java Archive (JAR) [27] format is the next step of the analyzer. This is a format that is widely accepted by the call graph generator used at later stages of the analyzer.

In order to package the repositories into JAR files, each project is built using its build management tool system. A tool specific command is ran by the analyzer on the parent *pom.xml* or *build.gradle* file of the project, which will create JARs for all of the modules that are part of the repository.

Generating Call Graphs

The generation process of CGs has been facilitated and enabled through the use of the FASTEN OPAL plugin [28]. With this tool, internal call graphs for both the project and its vulnerable dependencies were generated. The project CG was straight-forwardly constructed by giving the repository JAR file as input. Whereas for the vulnerable dependency, its artifacts were downloaded from Maven Central through the plugin code and given as input in a similar manner to the generator. The 2 individual graphs were merged into a single one, containing the method calls between the project and the dependency, on top of the internal calls within the 2 packages.

In addition, Rapid Type Analysis (RTA) [29] was used as a construction algorithm for both CGs. This algorithm is faster and better performing than its Class Hierarchy Analysis (CHA) counterpart, which also has an implementation provided by the plugin.

Graph Path Inspection

An adapted version of Breadth First Search (BFS) [30] traversal algorithm has been run on the merged graph in order to determine which vulnerable methods have been called from the project. The graph is traversed individually for each of the vulnerable functions of the dependency, using these as starting points. The paths from these methods in the direction of the dependant project are traversed and inspected. Each node found traversing the path is checked for containing a URI that

includes the repository name. If a node of this type is found, a vulnerability impact point on the project is discovered and added to a global set of points. An impact point is defined to be the last method in the project that called a function from the vulnerable dependency, which internally chain-called one of the vulnerable methods.

An impact point structure stores the caller method URI in the dependant project. Additionally, it stores the calling and origin vulnerable method URIs (that is, the function corresponding to the starting point of traversal) in the dependency. The set of impact points returned by the algorithm is logged to an output file on a per project and per vulnerability basis. As a result, the file will contain the 3 properties encapsulated by the impact points, together with the CG path traces.

4.4 Running the Analyzer and Verifying Results

The analyzer found a total of 564 distinct recently active projects that were vulnerable on the package-level. After method-level analysis was performed on this found subset, a total of 86 different CVE files dispersed over 62 unique repositories were discovered.

The result files have been verified to determine the correctness of the analysis. The verification process included inspecting whether:

- The mapping between the CVE and the methods from the dependency which are labeled as vulnerable is correct.
- The affected version range given by the CVE includes the version of the dependency.
- The resulting method calls between the project and the vulnerable dependency are correct.

Inspection further narrowed the number of results to 25 files coming from 24 distinct projects. These files were linked to 4 different CVEs: CVE-2019-14379 in the *jackson-databind* package [31], HTTPCLIENT-1803 affecting *Apache httpclient* [32], CVE-2017-9096 presenting security risks in *iText* [33] and lastly, CVE-2016-6797 in *Apache Tomcat* [34]. Table 1 illustrates a breakdown of the described vulnerabilities, including the number of appearances of each.

Table 1: Table containing the types of vulnerabilities found from running the analyzer and manually verifying the results

Vulnerability ID	Severity Level	No. appearances
CVE-2019-14379	CRITICAL	10
HTTPCLIENT-1803	HIGH	13
CVE-2017-9096	HIGH	1
CVE-2016-6797	HIGH	1

4.5 Pull Request Template

A GitHub PR has been manually opened for each of the 25 resulting method output files. A common template of the PR has been created, in order to standardize its structure and facilitate the opening process respectively. This template includes information such as:

- Elementary exposure information: linked CVE, vulnerability severity level and dependency release notes.
- Backwards compatibility of the bumped version with the current repository source code.
- Number of functions in the project calling vulnerable methods.
- A table listing the impact points of the vulnerability in the project.

4.6 Results

Out of the 25 opened Pull Requests, 4 (16%) have had developers react: 3 have been merged, while the other is still open. An overview of the status of the PRs can be seen in the chart of Figure 3.

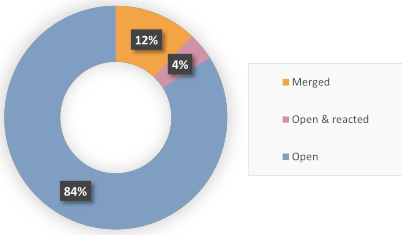

Figure 3: Chart breaking down the status of the PRs

Additionally, particularities of the repositories that reacted to the vulnerability fixes, such as popularity (given by number of stars and forks), number of contributors, past experience with dependency management tools, status of the PR in the repository and the found CVE are given in Table 2.

It is of interest to compare the merge times of the 3 PRs with the past merge times of older dependency related PRs of each repository. Through these comparisons, a measurement of PR prioritization can be quantified. To accomplish this, the mean and median merge times of older PRs were calculated, as well as the ratio of merged PRs (percentage of PRs in merged state, out of 3 possible states: open, closed, merged). Table 3 provides an overview of the aforementioned metrics for each of the 4 projects that reacted (repository #2 has also been included in the statistics although no merging was recorded, as the developers have indicated their will to merge the PR in the near future).

Repository labeled #1 had a recorded merge time for the PR opened by this study 30 times faster than the computed average and 6 times faster when set side by side to the median. This repository also had all of its dependency related PRs merged into the main branch. On the other hand, #2 merged 50% of all dependency related PRs in an average of roughly 8 days and median of 2. For project #3, the recorded time was equal to both the average and the median times. It had one PR that was merged, while the other was closed. Lastly, repository #4 had a slightly slower response than the average and median, whilst having the ratio between merged and open/closed PRs roughly evenly split.

Moreover, the developers of the 24 notified projects have been asked to complete a survey. At the moment, only #1 and #2 have filled it in. The survey included 6 statements, with

Project	Status	Stars	Forks	Contributors	Uses Dependabot	Affected by
#1	Merged	49	38	15	No	CVE-2019-14379
#2	Open & reacted	34	49	108	Yes	HTTPCLIENT-1803
#3	Merged	11	19	9	No	CVE-2019-14379
#4	Merged	144	57	24	Yes	HTTPCLIENT-1803

Table 2: Information about the reacting repositories' backgrounds, status of PRs and found vulnerabilities.

Project	No. dependency PRs	PR merge time			Ratio merged dependency PRs
		Recorded	Average	Median	
#1	7	1d	30.8d	6.5d	100%
#2	10	-	7.8d	2d	50%
#3	2	1d	1d	1d	50%
#4	11	22d	15.1d	1d	54.5%

Table 3: Statistics of dependency related Pull Requests in the projects that reacted to the PRs opened by this study.

the aim of investigating the effects of the provided method-level data on their prioritization and facilitation of fixing the vulnerability.

Statement 1: *I was aware of the vulnerability affecting my project before being informed by the Pull Request.*

Both of the repositories have answered that they were not aware of the notified vulnerability. Interestingly, even though #2 had a PR opened by Dependabot on the same vulnerable dependency (which probably included this security update), the information of this exposure did not reach the developers through the means of the package-level tool.

Statement 2: *I was convinced by the provided method call data that the vulnerability indeed affects my project.*

Project #1 has **strongly agreed** with the statement posed. The provided fine-grained data, consisting of more than 10 method calls between their project and the vulnerable dependency functions has thus contributed to their fast decision of merging the PR within 10 minutes. Moreover, #2 has also **agreed** with this statement. Even though this repository did not merge the PR, the given information (consisting of 1 method call) has persuaded them into giving attention to the exposure (which was shown by their action of labeling the PR within 1 day to elicit extra consideration from other developers).

Statement 3: *I plan on merging the PR in the near future.*

Both projects have responded positively to this statement. Project #1 confirmed this statement with their actions, whereas #2 still had the PR in an open state. The main reason why they did not merge is that the bumped version of the vulnerable dependency in the created PR was not backwards compatible with their source code. This version was a minor release in the same major as the previous one, but updating caused several tests to fail during the Continuous Integration (CI) pipeline run. This means that extra developer effort is required in order to identify, locate and fix the reasons why the tests break. However, the developers of this repository indicated their will to resolve the issues and merge the PR in the near future.

Statement 4: *The provided method call information has made my process of dealing with the vulnerable dependency easier (when comparing with past experience).*

Repository #1 **strongly agreed** with the statement. It can be noted that using the provided data, they were able to locate the vulnerability points of impact in their source code. Noteworthy to mention is that this project did not use any dependency management tool in the past and thus the maintenance process has been done on a manual basis (with a history of 7 PRs related to package uplifts/downlifts). On the other hand, #2's response was **neutral**, arguing that the provided information did not have any effect on facilitating their vulnerability fix process. This repository did use Dependabot and could thus provide a comparison between fixing exposures with help from package-level and from method-level data respectively.

Statement 5: *I have given priority to the task of fixing the vulnerability over other project tasks that are yet to be completed.*

The two repositories responded **Yes** to this statement. Based on the fact that both projects have been convinced by the provided information that their repositories are at risk, it can be said that incorporating this type of data in the PRs has had an effect on them prioritizing the process of fixing the vulnerability.

Statement 6: *I would like the idea to receive this kind of method information in future vulnerable dependency Pull Request descriptions.*

Developers of #1 and #2 indicated their interest in receiving this type of data in future security related PRs. This likely means that they find the provided information useful, believing that having it at their disposal could provide aidance in fixing the vulnerabilities.

Summary of RQ: Although not having enough collected data at the moment to be conclusive, it has been observed that providing the developers the set of method calls between their projects and vulnerable dependencies makes them prioritize the task of fixing vulnerabilities. Moreover, developers indicate that their security fix process is to an extent facilitated by the provided data.

5 Responsible Research

In this study, developers of projects containing vulnerable dependencies are notified of the exposures found in their re-

liances. These vulnerabilities are already publicly disclosed for the dependency project and have had patching versions deployed. As a result, each found vulnerability is not novel (a published CVE for it exists) and it could have been discovered by any dependency maintenance tool running against the project or by any interested individual.

The main intention of advising the developers is to improve the safety and stability of the source code of their repositories, by opening Pull Requests that fix the vulnerabilities. Along with the benefits of the project, the users also have to gain from having an overall safer user experience. For example, the affected project could have stored sensitive user credentials or data with the help of the vulnerable dependency. In this case, the exposure would have had an impact on handling sensitive user data and providing a fix for it would have thus been critical.

In addition, the data collected through the surveys has been processed and displayed in a completely anonymous manner in the paper. No references in this report have been made to the affected repositories, as they have been assigned a unique number that was used as a label. This eliminates the threat of disseminating the project vulnerabilities to the public.

The methodology described in Section 3 clearly lays out the steps that have been taken in order to conduct this study, along with the main motivations behind each. The FASTEN Project database has been used as a data source for finding the project selection set and the vulnerable packages to look for as dependencies. The extraction process of these has been explained in Section 4.

Furthermore, analysing the selected projects on the package and method levels constitutes a large part of the study. To accomplish this, an analyzer has been implemented. A diagram depicting a high level overview of the system, as well as detailed implementation aspects of this analyzer have been given in the Experimental Setup Section. Moreover, this analyzer makes use of the OPAL plugin and of the Dependabot Script, which are both open-source projects available for use.

It is important to note that future studies employing the methodology described in this paper will most probably not work with the same set of projects. This is because the used repositories were extracted from the FASTEN database, which is constantly being updated. Besides this aspect, the methodology will work equally well on a different set of retrieved projects.

6 Discussion

The provided method-level information has been observed to have a positive effect on the developers' prioritization and facilitation of fixing vulnerabilities. Observations were made with aid from collected survey data and comparisons between recorded PR merge times and past dependency related PRs statistics. However, a couple more associations can be listed out between the metrics that were previously presented.

Firstly, projects #2, #3 and #4 have a lower ratio of merged dependency related PRs than #1. These are all projects having their recorded PR merging time greater than or equal to their computed past average and median times. Similarly, a correlation between the high ratio of merged dependency PRs and the vulnerability fix prioritization can be observed for #1.

Furthermore, the merging times could be differentiated upon the vulnerability severity type affecting the projects. Repositories #1 and #3, which both merged the PRs within 1 day, were affected by a *CRITICAL* level vulnerability. In contrast, the remaining repositories were at risk due to an exposure marked as *HIGH*, incurring longer merge times. This finding contrasts with the observations of Alfadel et al. [35], which found that Dependabot PR merging times were not affected by vulnerability severity level. As such, there is a possibility that the provided extra method-level information makes the developers more conscientious towards the risk imposed by a vulnerability, as they can see that their projects are actually affected.

Interestingly, based on the collected data until this moment, no correlation between project popularity and security fix prioritization has been recorded. Neither did the number of contributors of a repository play a role in the fast merge times.

Only 12% of the developers have merged the PRs until this time. For the rest of the repositories, the factors that could have contributed to the lack of reactions have been analyzed. As a result, the following motives have been identified:

1. **Development activity follows an irregular trend.** All notified repositories are part of Open Source Software. A study on commit frequency distribution in OSS [36] revealed that the 95th percentile of median time interval between 2 commits of the same author is 51.4 days. This indicates that development activity often times goes through larger pauses in these types of projects.
2. **Developers did not react to Dependabot PRs.** It has been observed that many of the notified repositories have older Dependabot PRs that are still in an open state. Interestingly, some of these are even opened to update the vulnerable dependency found by this study.
3. **Automated PR checks following dependency uplift were not successful.** Part of the projects have experienced build failures following the dependency version uplifts (due to backwards compatibility issues). As a result, overhead is likely incurred in these repositories, as extra developer effort is required in identifying and fixing the failures. This effort in testing a new version of a dependency has been linked to cause maintainers troubles in updating to it [37].
4. **Bumped dependency version is not compatible with the project requirements.** A project having an older Dependabot PR that uplifted the dependency found vulnerable also by this study was closed because of the bumped version's functional incompatibility with the project's requirements. The following comment was left by one of the developers in the closed PR: *"This is incompatible with our encoding requirements"*.

7 Conclusions and Future Work

This paper presented a qualitative study on how developers react to method-level information provided in GitHub Pull Request descriptions. Important factors that were analyzed were the effects of this type of data on the developers' priori-

tization and facilitation of the vulnerability fix process in the scope of their affected repositories.

Concretely, 24 projects were notified through Pull Requests of the exposures found in their source code, out of which 4 reacted (3 merging them and 1 labeling it). Developer behaviour data was then collected through surveys. In total 2 developers have filled in the provided survey. These were maintainers of a project which merged the Pull Request within 1 day and of a repository that labeled the Pull Request and indicated their will of merging it in the near future.

Examining the background of the repositories that responded and analysing their answers, it has been observed that the provided method-level data has had a positive effect on the prioritization of fixing vulnerabilities in their repositories. Moreover, the maintainers have found the provided data useful, mentioning that they would be interested in receiving this type of information in future security related Pull Requests. It has also been observed that the fine-grained data has facilitated the developers' exposure fix process to a certain degree.

Important to note is that insufficient data has been collected, making it hard to support the observations and be conclusive. Nonetheless, it is expected that more results will emerge in the near future from the currently non-responding repositories. The lack of reactions is assumed to be caused by all target projects being part of Open-Source Software, which typically have developers that contribute and monitor projects sporadically.

A study improvement would be to expand the set of target projects. Enlarging the set would give the possibility to scan more repositories for vulnerabilities, which in turn would lead to finding more affected projects that can be ultimately notified. Furthermore, a more reliable metric for labeling projects as active could be used. This study has used the last update date of a repository (not older than 4 months) as a distinguishable factor. A better suggested metric would be the commit frequency distribution of a repository, as it covers the overall development activity trend.

Future work can expand on the foundations laid by the observations discovered through this study. By means of gathering more developer responses regarding their reactions to fine-grained data provided in GitHub Pull Requests, definitive conclusions will be reached with respect to the prioritization and facilitation of fixing vulnerabilities through method-level data aidance.

References

[1] R. Cox, "Surviving Software Dependencies," *Commun. ACM*, vol. 62, no. 9, p. 36–43, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3347446

[2] T. LaToza, G. Venolia, and R. Deline, "Maintaining mental models: A study of developer work habits," vol. 2006, 01 2006, pp. 492–501.

[3] V. Piantadosi, S. Scalabrino, and R. Oliveto, "Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat," 11 2019.

[4] J. Hejderup, A. v. Deursen, and G. Gousios, "Software Ecosystem Call Graph for Dependency Management," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, May 2018, pp. 101–104.

[5] "Dependabot," Accessed on: Jun. 21, 2021. [Online]. Available: https://dependabot.com/

[6] D.-L. Vu, *A Qualitative Study of Dependency Management and Its Security Implications (To be appear in ACM CCS 2020)*, 08 2020.

[7] P. Boldi and G. Gousios, "Fine-Grained Network Analysis for Modern Software Ecosystems," *ACM Transactions on Internet Technology*, vol. 21, no. 1, pp. 1:1–1:14, Dec. 2020. [Online]. Available: http://doi.org/10.1145/3418209

[8] S. Foster, "Vulnerabilities Definition: Top 10 Software Vulnerabilities," Jul. 2020, Accessed on: Jun. 21, 2021. [Online]. Available: https://www.perforce.com/blog/kw/common-software-vulnerabilities

[9] GitHub Docs, "About coordinated disclosure of security vulnerabilities," Accessed on: Jun. 21, 2021. [Online]. Available: https://docs.github.com/en/code-security/security-advisories/about-coordinated-disclosure-of-security-vulnerabilities#about-disclosing-vulnerabilities-in-the-industry

[10] National Vulnerability Database, "General Information," Accessed on: Jun. 21, 2021. [Online]. Available: https://nvd.nist.gov/general

[11] GitHub Docs, "About github security advisories," Accessed on: Jun. 21, 2021. [Online]. Available: https://docs.github.com/en/code-security/security-advisories/about-github-security-advisories

[12] npm, "npm Security Advisory," Accessed on: Jun. 21, 2021. [Online]. Available: https://www.npmjs.com/advisories

[13] A. T. Tunggal, "What is CVE? Common Vulnerabilities and Exposures Explained," Mar. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://www.upguard.com/blog/cve

[14] National Vulnerability Database, "Common Vulnerability Scoring System," Accessed on: Jun. 21, 2021. [Online]. Available: https://nvd.nist.gov/vuln-metrics/cvss

[15] GitHub Docs, "About Dependabot security updates," Accessed on: Jun. 21, 2021. [Online]. Available: https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/about-dependabot-security-updates#about-dependabot-security-updates

[16] FASTEN Project, "Fasten Project," Jun. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://www.fasten-project.eu/view/Main/

[17] "What is a Maven Repository?" Accessed on: Jun. 21, 2021. [Online]. Available: https://www.tutorialspoint.com/maven/maven_repositories.htm

[18] Apache Maven Project, "Welcome to Apache Maven," Jun. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://maven.apache.org/

[19] "Gradle Build Tool," Accessed on: Jun. 21, 2021. [Online]. Available: https://gradle.org/

[20] Apache Maven Project, "Introduction to the POM," Jun. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html#what-is-a-pom

[21] S. McLeod, "Likert Scale Definition, Examples and Analysis," 2019, Accessed on: Jun. 21, 2021. [Online]. Available: https://www.simplypsychology.org/likert-scale.html

[22] M. Sokolov, "Metadata Database Schema," Feb. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/fasten-project/fasten/wiki/Metadata-Database-Schema

[23] E. Lanzini, "Vulnerability Analyzer," Mar. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/fasten-project/fasten/wiki/Vulnerability-Analyzer

[24] ——, "Vulnerability Object Definition," Mar. 2021, Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/fasten-project/fasten/wiki/Vulnerability-Analyzer#vulnerability-object-definition

[25] "Package URLs," Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/package-url/purl-spec#readme

[26] Dependabot, "Dependabot Script," Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/dependabot/dependabot-script

[27] "What is a JAR file?" Accessed on: Jun. 21, 2021. [Online]. Available: https://docs.fileformat.com/programming/jar/#what-is-a-jar-file

[28] FASTEN Project, "JavaCG-OPAL," Accessed on: Jun. 21, 2021. [Online]. Available: https://github.com/fasten-project/fasten/tree/develop/analyzer/javacg-opal

[29] B. Holland, "Call Graph Construction Algorithms Explained," Mar. 2016, Accessed on: Jun. 21, 2021. [Online]. Available: https://ben-holland.com/call-graph-construction-algorithms-explained/

[30] "Breadth First Search or BFS for a Graph," Dec. 2020, Accessed on: Jun. 21, 2021. [Online]. Available: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

[31] National Vulnerability Database, "CVE-2019-14379 Detail," Accessed on: Jun. 21, 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2019-14379

[32] "HTTPCLIENT-1803: Malformed path not handled well," Accessed on: Jun. 21, 2021. [Online]. Available: https://issues.apache.org/jira/browse/HTTPCLIENT-1803

[33] National Vulnerability Database, "CVE-2017-9096 Detail," Accessed on: Jun. 21, 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-9096

[34] ——, "CVE-2016-6797 Detail," Accessed on: Jun. 21, 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016-6797

[35] M. Alfadel, D. Costa, E. Shihab, and M. Mkhallalati, "On the Use of Dependabot Security Pull Requests," 02 2021.

[36] C. Kolassa, D. Riehle, and M. Salim, "The Empirical Commit Frequency Distribution of Open Source Projects," *Proceedings of the 9th International Symposium on Open Collaboration, WikiSym + OpenSym 2013*, 08 2014.

[37] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring Dependency Freshness in Software Systems," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, 2015, pp. 109–118.