The State of Open-Source Security: Supply Chain Attacks in NodeJS Projects and the Role of GitHub Dependabot

Lorenzo Bartolini
lorenzo.bartolini8@edu.unifi.it
University of Florence
Italy

Angelo Rosa
angelo.rosa@edu.unifi.it
University of Florence
Italy

***Abstract*—Supply chain attacks are becoming increasingly common as the days pass. Thus, we need a study on this subject that aims to highlight the most critical attack vectors. In our study, we identified the dependencies of projects inside the *npm* ecosystem as a big vector of vulnerabilities, remembering that a vulnerability is not only a software bug, but it can also be voluntarily introduced by some bad actors. With our work, we aim to understand how many repositories contain vulnerable packages, how many of them present critical vulnerabilities, how *CVEs* are distributed and a classification by the type of vulnerabilities present.**

***Index Terms*—Supply Chain Attacks, NPM Packages, CVE, CWE, Vulnerabilities, GitHub, Scraping**

I. Introduction

According to GitHub organization, the number of malwares installed via malicious packages is increasing every year for any language and for any official package manager [1]. In our research, we want to focus on one of the most used languages (`JavaScript`) and its technology (`nodejs`), which is very used not only for client side application but also for server side applications. Thus, we will not only take a look at Supply Chain Attacks and involved packages, which are becoming a trending attack, but we will also take a look at all vulnerable npm packages that are usually stored in a JSON format in the file *package.json* among the repositories.

Summarizing the work we have done, we used some public `REST API` provided by GitHub for searching among JavaScript repositories and their files. Then, we looked at `package.json` files and we divided our repository collection into six main batches, each one containing around 900 repositories. After this process, we analyzed packages accessing other vulnerability databases in order to obtain the list of `CVE` and `CWE`, eventually along with some `POCs` [2]. At this point, we obtained how many vulnerabilities affect each package and what they are. We proceed further with another analysis which allowed us to get all `CVE` used, and we classified each one into different categories. What has been described so far has allowed us to obtain some deep interesting statistical results which reflect the current status of the Open-Source Security. A strong attention has been put in making our code general and reusable for future analysis, the various scripts used for collecting such data can be easily extended for analyzing other package managers, other programming languages, and other dependency files, such as `PyPI`, `Go package manager` and so on.

Before digging into details, we wanted to underline the importance and the role of *Dependabot* used by GitHub. As its name says, the bot can be installed inside any repository to warn maintainers when a dependency becomes obsolete or a security issue occurs. It is very useful for updating all dependencies as it is also able to create pull requests or merge them automatically, avoiding security incidents. As our study will show, unfortunately, this technology is often ignored by maintainers rather than used to update packages avoiding potential security incidents.

This paper will be structured as follows. In Section II we will describe the background knowledge necessary to understand the rest of the paper. Inside it we will talk about Supply Chain Attacks and npm-Hack in particular. Next, in Section III we will explain how we reached our goals with a high level description of our code and the decisions made along. The results will be showed in Section IV. We will also discuss them in depth with numbers and figures to support our findings. Then in Section V we will explain what Dependabot is and its role with respect to Supply Chain Attacks. In the end, in Section VI we will wrap up our work.

This is the link to the repository containing the code used to scrape and extract all the data and information, https://github.com/LBartolini/ProgettoSAM.

II. Background

A. Supply Chain Attacks

According to MITRE [1]:

[1] https://github.blog/security/github-advisory-database-by-the-numbers-known-security-vulnerabilities-and-what-you-can-do-about-them/

[2] Proof Of Concept Scripts

> A Supply Chain Attack is an attack that allows the adversary to utilize implants or other vulnerabilities inserted before installation to infiltrate data, or manipulate information technology hardware, software, operating systems, peripherals (information technology products) or services at any point during the life cycle.

It is clear that Package managers can become a gold mine because hacking a single package inside their database allows for automatically spreading the attack effortlessly across a large number of projects and repositories. This is obvious since all users trust and rely on the official repository provided by the programming language and daily checked by maintainers. Also, infecting a particular version of a library is critical because projects rarely keep they're dependencies updated. To give a full background of the situation, we will mention 3 very famous cases that are outspread nowadays.

1) `xz-utils backdoor` [3] happened in 2024
2) `npm-hack` [4] happened in 2025
3) `log4shell` [5] happened in 2021

With this list, we want to underline the difference between a Supply Chain Attack and the presence of a critical vulnerability due to a human error. `XZ-utils backdoor` and `npm-hack` are 2 among the most famous Supply Chain Attacks attempts, while `log4shell` is a Java library widely used in the Java ecosystem for logging purposes. The last one suffered from a `Remote Command Execution (RCE)` vulnerability, which allowed the injection of an arbitrary command on a remote machine that was using the library. The result was a catastrophic 90% of cloud applications using log4j being vulnerable to RCE in a matter of seconds; a lot of those services were infected with a sophisticated trojan [6]. An interesting question immediately comes to mind:

> *How many are still vulnerable to a well known vulnerability such as log4shell in 2025?.*

Our study revolves around trying to answer this question focusing on vulnerabilites affecting Javascript NodeJS projects.

Coming back again to `xz-utils` hack, this one was a very complex attack which involved several attacking techniques from social engineering to cryptographic madness to deliver a well-made backdoor affecting `liblzma` library, used by a lot of other software. Luckily, the attack was discovered before malicious actors were able to deliver the infected package to the entire Linux community, but the method used is still under research. For such a complex attack, it has been hypothesized that some government agency could be involved, but this theory, like many others, has never been confirmed.

B. npm-Hack

The `npm-hack` involved two supply chain attacks that occurred in September 2025, affecting more than 100 packages. Even if slightly different, the purpose of both the first [7] and second [8] attacks was to steal private keys of developers stored locally or in the cloud, not only to steal sensitive data but also to auto-reproduce the malware by infecting other packages that the same developers maintain. The result was an auto-replicant trojan which involved very well-known packages downloaded weakly millions of times by all `javascript` developers around the world.

The involved packages still do not come with a provided CVE, but a common CVE name may be provided in the future for identifying this attack. In any case, each single package delivered by affected maintainers resulted in being vulnerable, containing malicious code. For example, `@ctrl/magnet-link` version `4.0.4` can be easily found in one of databases used by us: *https://osv.dev/vulnerability/MAL-2025-47133*

So, by providing the package name with its version to some of these databases, we were able to extract either CVE, GHSA or any other format for identifying the presence of the vulnerability. It is also important to underline that this attack did not need any interaction with users apart from installing the compromised package. This is because the infection phase happens at installation time, regardless of its usage in any JavaScript project. Thus, it is not important to check if any vulnerable package to any CVE is effectively used. For most cases, it is enough to find it in the `package.json`.

III. Study Design

The goal of our research will be achieved via scraping Open-Source Javascript projects on GitHub. Our focus is going to be towards the *npm* ecosystem looking for *package.json* files inside the repositories. To do so, we relied upon the GitHub Search API. The repositories to analyze were divided and filtered into 6 batches by number of stars. This way we were able to obtain a representative sample of all the repositories, ranging from less famous to more famous ones. The following are the star ranges used:

- Between `19` and `22` stars
- Between `50` and `65` stars
- Between `150` and `240` stars

[3] https://en.wikipedia.org/wiki/XZ_Utils_backdoor

[4] https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/

[5] https://en.wikipedia.org/wiki/Log4Shell

[6] https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell

[7] https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/

[8] https://snyk.io/blog/embedded-malicious-code-in-tinycolor-and-ngx-bootstrap-releases-on-npm/

Filter Name	Description
language	Filters repository by language. We used *javascript*
size	Used to limit the size of the repositories. We fixed an upper threshold of 100MB to reduce both computational power and number of API calls
stars	Filters by stars. Accepts star ranges (19..22, 50..65, etc) or thresholds (>1800, etc)
pushed	Filters repositories that pushed their last commit in the range provided. We used this filter to consider only active repositories that pushed at most one month ago
per_page	Used to select how many results to show for each API call. The maximum is 100
page	Used to select the page for the specific request

TABLE I
GITHUB SEARCH FILTERS

- Between `240` and `450` stars
- Between `450` and `999` stars
- Greater than `1800` stars

The API call has been constructed as follows using the filters described in Table I:

```
https://api.github.com/search/repositories
?q=language:{language}+size:<{size_up_limit}
+stars:{stars}+archived:=false
+pushed:{last_commit_pushed_after}..{today}
&per_page={per_page}&page={page}
```

After obtaining the fitting repositories, the next step we had to perform was creating a set for each repository. Such set should contain all the dependencies, in the format of `(dependency name, version)`, found inside all package.json files within the repository. To achieve this, we further scraped the repositories, one by one, through the GitHub API, looking for the searched file, package.json. After finding the files, we extracted the dependencies using a Regex and putting the result in a *set* to automatically remove duplicates. Dependencies inside package.json are exctracted from three places: *dependencies*, *devDependencies* and *bundleDependencies*. Once we have a complete set of all the dependencies with the associated versions, we queried some services asking for `Common Vulnerabilities and Exposure (CVE)`[9] and other useful information. The results from each service were then merged together to increase accuracy.

The services used are:

- *Shodan*[10], for CVE, CVSS, EPSS score
- *OSDev*[11], for CVE, CVSS, GHSA, Severity
- *Deps.dev*[12], for GHSA
- *Sploitus*[13], for Proof-of-Concept (POC)

[9]https://www.cve.org/
[10]https://cvedb.shodan.io/
[11]https://osv.dev/
[12]https://deps.dev/
[13]https://sploitus.com/

Column Name	Description	Example
repository	Name of the repository in the format Owner/RepoName	chartjs/Chart.js
number_of_commits	Total number of commits	4574
number_of_contributors	Number of contributors	497
bytes_of_code	Total number of bytes of Javascript source code files	1211966
star_range	Range of stars used during scraping	>1800
star	Precise number of stars	66571
latest_push	Date and Time of latest push on GitHub	2025-09-22T15:27:57Z
total_dependencies	Total number of unique dependencies found in package.json files inside the repository	75
number_of_vulnerabilities	Number of vulnerabilities found across dependencies	7
ghsa	GHSA strings separated by comma	GHSA-v78c-4p63-2j6c, ...
cve_strings	CVE strings separated by comma	CVE-2024-9506, ...
cvss	CVSS scores associated to CVEs	6.1, ...
severity	Severity associated to CVEs	HIGH, ...
sploitus_number_of_pocs	Number of publicly available Proof-of-Concepts associated to CVEs	0

TABLE II
CSV COLUMNS DESCRIPTION

The CVE and GHSA strings represent known vulnerabilities associated to a package, service or software in general. CVE is more famous and internationally recognized while GHSA is managed by GitHub and is the one used by Dependabot. In our study we focused on CVEs since they are the most used. The `CVSS Score` and the `Severity` are attributes related to a software vulnerability. Both express a technical severity, the `CVSS` is a score from 0 to 10 where 10 is the most critical; the Severity instead can be `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`. Both these attributes were used inside our study. Finally, with the term `Proof-of-Concept (POC)` we refer to a small piece of code that shows a possible exploitation of a vulnerability. It can be used as a starting point for deeper exploits. We looked for POCs because it is helpful to know if an exploit for a vulnerability is publicly available or not, especially given that even less skilled attackers can use them.

The total number of repositories collected this way is 5375. Then we removed the ones not containing any package.json. In the end, the number of repositories used for the analysis is *4429*.

The generated CSV contains all the relevant information about the repositories. The column description of the CSV are shown in Table II.

After the scraping was finished, the last work done was on CVEs. We collected all the CVEs across all repositories. We then further scraped public databases to categorize all CVEs in several Macrogroups. Moreover we were able to

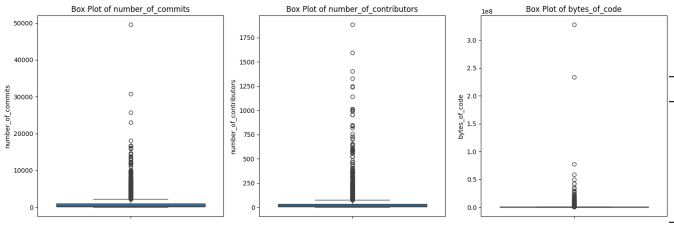

Fig. 1.

TABLE III
DESCRIPTIVE STATISTICS FOR DEMOGRAPHIC METRICS

Statistic	number_of_commits	number_of_contributors	bytes_of_code
Standard Dev.	1968.00824	98.869028	6.634958e+06
Mean	985.79	39.19	8.727643e+05
Min	1	1	0
50% (Median)	375	13	1.089830e+05
Max	49,542	1,881	3.275376e+08

collect `Common Weakness Enumeration (CWE)` for the CVEs in our hands using the MITRE API[14]. The main difference between CVE and CWE is that CVE express a specific vulnerability of a software component, on the other hand CWE represents an intrinsic weakness in the code that can lead to an actual vulnerability, this means that we can find more than one CWE associated to a single CVE. To sum up, CWEs are the fundamental cause of vulnerabilities and CVEs are the instances of such vulnerabilities in specific components.

Lastly, we gathered all the EPSS Scores and EPSS Rankings for each CVE. The EPSS score is a value that represents the probability that a particular CVE is exploited *in the wild* in the next 30 days. This score, together with CVSS, is crucial for CyberSecurity Engineers to understand the risks associated to the packages their software use. The EPSS Ranking represents the percentile, inside the EPSS system, associated to a specific CVE. Meaning that a ranking of 0.95 means that it is in the Top 5% of the highest EPSS score.

In the following section we will show the statistical results showing many interesting facts about Security in Open-Source projects.

IV. RESULTS AND DISCUSSION

The results obtained will be described and discussed in this section. We divided them into four subsection to better discuss each topic.

The topics we will discuss are: *Clean vs Vulnerable Repositories*, *Analysis of Vulnerable Repositories*, *Study of CVEs/CWEs* and *EPSS System*.

Before diving into them, we take a look at the demographics of our dataset.

A. Demographics

Figure 1 shows the distribution of analyzed repositories using the following features.

- *number of commits*
- *number of contributors*
- *bytes of code*

About `number of commits`, the median is very close to the bottom of the scale, suggesting that most repositories have a relatively low number of commits. The box is quite compressed, indicating that the majority of repositories fall within a narrow range of commit counts and there are many outliers, some reaching up to 50,000 commits, showing that a few repositories are extremely active compared to the majority. A similar pattern can be seen for the `number of contributors` as the median is low but there are numerous outliers and only few repositories reach more than 1700 contributors. The median and box are both clustered near zero, while a few extreme outliers reach up to hundreds of millions of bytes; thus, most repositories are relatively small in terms of codebase size, but some are exceptionally large. These statistics obtained were quite expected.

In Table III we can notice that for the `number of commits`, the median (375) is much lower than the mean (986), indicating a right-skewed distribution. Similarly, for `number of contributors` the median (13) is far below the mean (39), showing that most repositories involve small teams, with some exceptions. `Bytes of codes`: the mean (about 872Kb) is vastly greater than the median (about 109Kb), and the maximum value (about 327 million bytes) highlights the presence of a few very large codebases. This again reflects a highly skewed distribution.

We have also conducted a `shapiro-walk` test for normality. Here will follow the results:

Column: number_of_commits
Test statistic: 0.4529
p-value: 0.0000
The column does not appear to be normally distributed (we reject the null hypothesis).

Column: number_of_contributors
Test statistic: 0.3508
p-value: 0.0000
The column does not appear to be normally distributed (we reject the null hypothesis).

Column: bytes_of_code
Test statistic: 0.0728
p-value: 0.0000
The column does not appear to be normally distributed (we reject the null hypothesis).

[14]https://cveawg.mitre.org/api/cve/cve-id

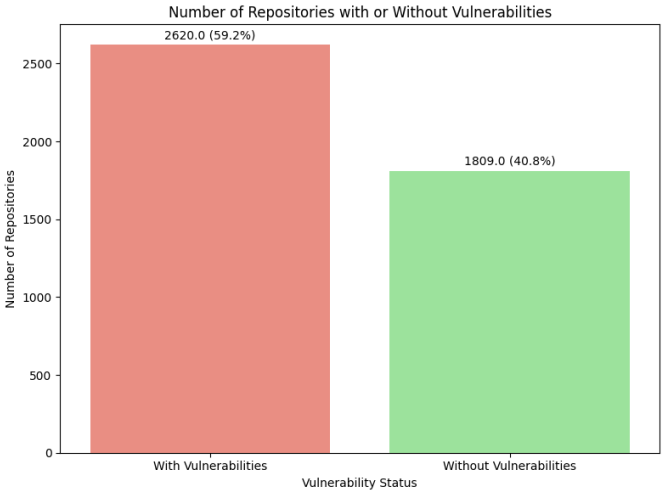

Fig. 2.

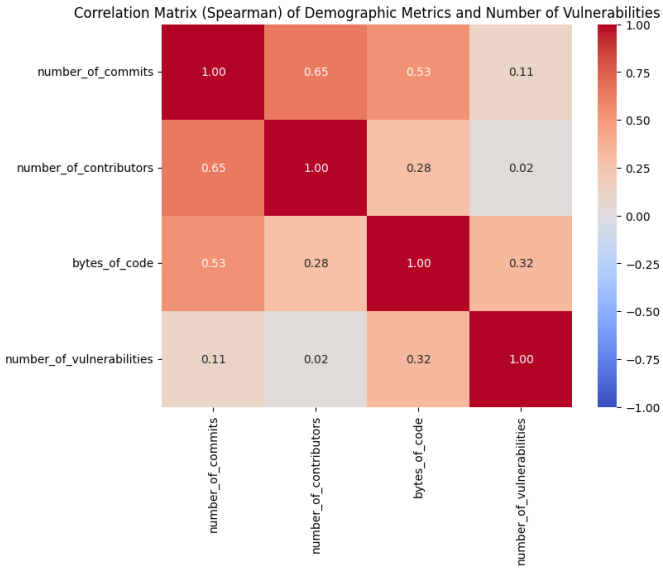

Fig. 3.

B. Clean vs Vulnerable Repositories

Our first analysis focuses on all the repositories scraped before. We wanted to show how many repositories were afflicted by vulnerabilities. Initially, we do not concentrate on a specific type of Severity, instead we will just look at all vulnerabilities to get a first idea of the state of public repositories.

Figure 2 shows the percentage of repositories that are affected by at least one vulnerability. Those repositories will be called Vulnerable Repositories in the next paragraphs. What we can see is that almost 60% (2620) of all the repositories (total 4429) analyzed are considerable Vulnerable, meaning that they include flawed packages in their code. This is very concerning since these are publicly available software and are probably running or being used by many users. They can serve as a starting point for deeper and more sophisticated attacks.

Now we want to analyze the Spear-Man correlation matrix obtained between `demographics` and the `number of vulnerabilities` (Figure 3):

- number_of_commits vs number_of_vulnerabilities (0.11): There is a very weak positive correlation between the number of commits and the number of vulnerabilities. This suggests that as the number of commits increases, the number of vulnerabilities tends to increase slightly, but the relationship is very weak.
- number_of_contributors vs number_of_vulnerabilities (0.02): The correlation between the number of contributors and the number of vulnerabilities is almost zero. This indicates that there does not appear to be any significant monotonic linear relationship between these two metrics.
- bytes_of_code vs number_of_vulnerabilities (0.32): There is a weak-to-moderate positive correlation between code size (in bytes) and the number of vulnerabilities. This result is more meaningful compared to the other demographic metrics and suggests that larger repositories tend to have a higher number of vulnerabilities.

The correlations among the demographic metrics themselves (`number of commits`, `number of contributors`, `bytes of code`) are stronger, as expected (e.g., more commits and contributors can lead to more code).

The Spearman correlation analysis suggests that among the demographic metrics considered, code size `bytes_of_code` has the strongest (though still weak-to-moderate) positive correlation with the total number of vulnerabilities. The number of commits and, in particular, the number of contributors show a very weak or nonexistent correlation with vulnerability. This may indicate that project size is a slightly more influential factor on the number of vulnerabilities than development activity or the number of people involved, at least in terms of monotonic linear relationships.

The next question we tried to answer was:

> *What is the distribution of vulnerable repositories when looking at the stars they received on GitHub?*

The answer to this question can be seen in Figure 4. The percentage of vulnerable repositories grows w.r.t the number of stars received. This is even more concerning because popular and famous projects are more likely to be used and trusted by many people. In particular we want to stress out how repositories with more than 1800 stars contain more vulnerable repositories that the average computed before. In fact they reached a worrying 64% of vulnerable repositories. This shows that even big projects can be susceptible to these kind of attacks.

Fig. 4.

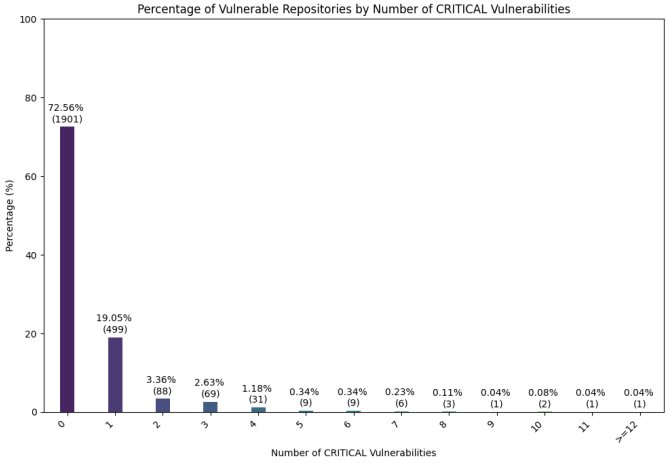

Fig. 7.

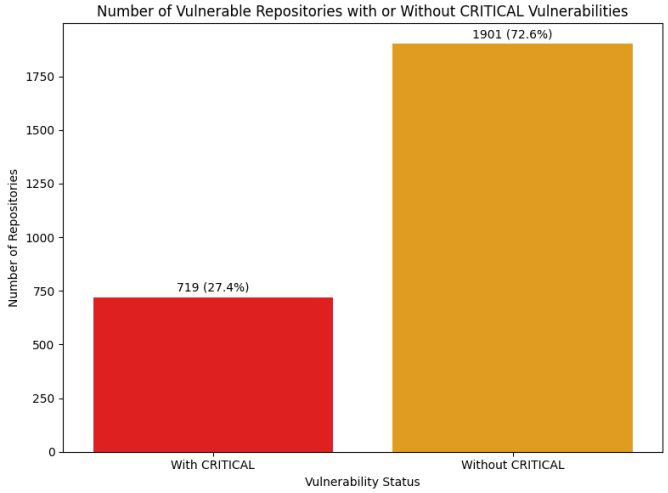

Fig. 5.

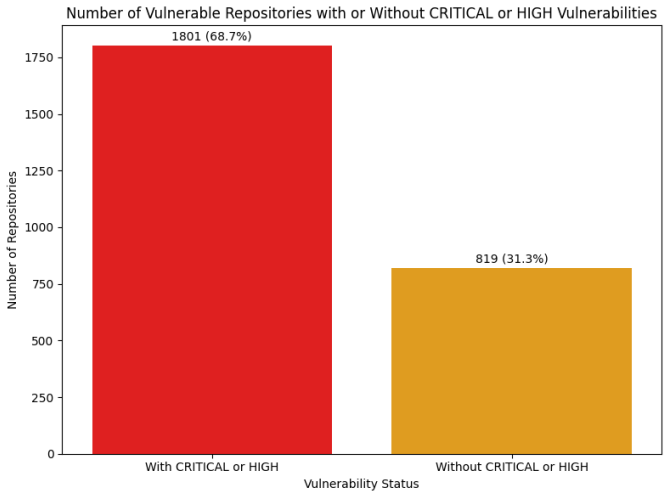

Fig. 8.

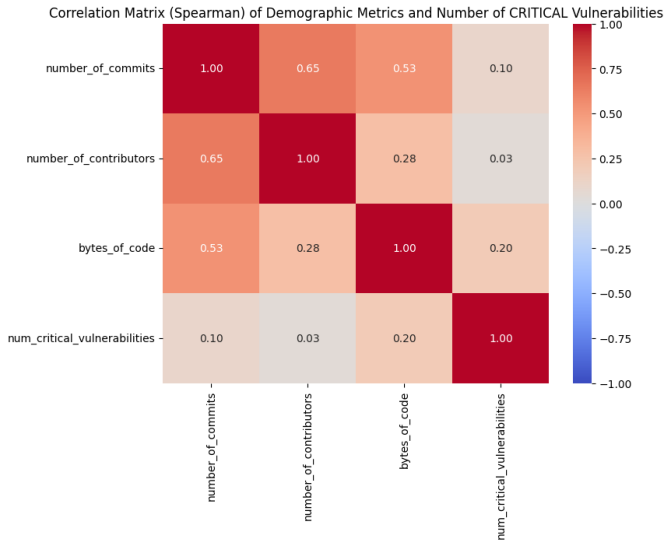

Fig. 6.

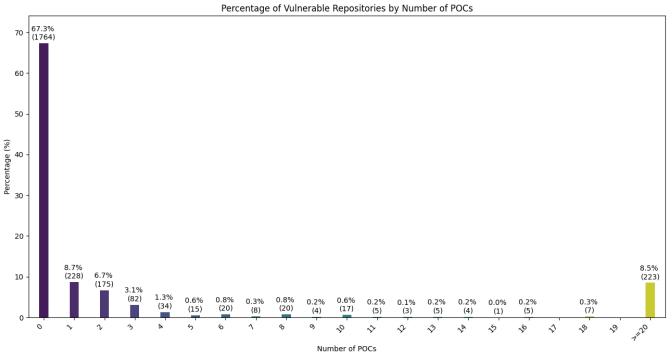

Fig. 9.

C. Analysis of Vulnerable Repositories

Up to now, we have gained an overview of repositories that contained at least one public CVE among their dependencies. The results were interesting, but what we will see in this section could be even more. Now we will talk about critical vulnerabilities only. As many users involved in cybersecurity will know, having a CVE in a package does not mean that the exploitability is easy or, in some cases, even possible. For exploiting some vulnerabilities, some constraints must be respected, and that's why generally a `CVSS` score is used, indicating `Confidentiality`, `Integrity` and `Availability` along with other parameters which help to understand the criticality of the situation. For example, a package containing a vulnerability `MEDIUM` could have a lower impact on itself, but its `CVSS` can be assigned based on the fact that it is harder or easier to exploit. We will discuss better `CVSS` and `CWE` in the next section. For the moment, it is important to understand that a package with medium vulnerability can escalate to a higher level of criticality in conjunction with the exploitation of other vulnerabilities, but alone, it's difficult to achieve a critical bug. However, this is different for packages that have already been marked containing vulnerabilities `Critical` because the worst ones require less or no user interaction to be exploited.

This is why we are interested in going into details and analyzing only packages containing at least a `CRITICAL` vulnerability. The result, as shown in Figure 5, is not better than the more general one analyzed in the previous section. As we can see, 27,4% among all vulnerable repositories contain at least one `CRITICAL` vulnerability. If this result seems to be irrelevant, think that this probably means that a `Remote Command Execution` or `Command Injection` is most likely possible, leading to the total compromising of a server-side application without a developer even being able to prevent it.

Now we take a look at the Spearman correlation matrix between `demographics` metrics and `number of critical vulnerabilities` (Figure 6):

- *number_of_commits* vs *num_critical_vulnerabilities* (0.10): There is a very weak positive correlation between the number of commits and the number of CRITICAL vulnerabilities. Similar to the case of total vulnerabilities, the effect is minimal.
- *number_of_contributors* vs *num_critical_vulnerabilities* (0.03): The correlation between the number of contributors and the number of CRITICAL vulnerabilities is practically nonexistent.
- *bytes_of_code* vs *num_critical_vulnerabilities* (0.20): There is a weak positive correlation between code size and the number of CRITICAL vulnerabilities. This correlation is weaker than that observed with the total number of vulnerabilities but still indicates a tendency for larger repositories to have more CRITICAL vulnerabilities.

Spearman correlation matrix analysis for critical vulnerabilities is very similar to the correlation results obtained for the total number of vulnerabilities.

Figure 7 shows the distribution of vulnerable repositories by the number of critical vulnerabilities found in `package.json` files. As expected, a good number of repositories contain no vital bugs. However, we found that some repositories not only contain a critical vulnerability (499 packages, 19.05%) but also a not insignificant number of them contain more than one package affected by a critical vulnerability. This is particularly interesting as we reach a single repository containing more than 12 critical bugs within its dependencies! Also, as already discussed, this leaves more attack vectors and way more possibilities to exploit the target, considering that these repositories are usually libraries or frameworks, and if imported by other projects, they will likely infect other repositories recursively.

When analyzing `CRITICAL` plus `HIGH` vulnerabilities, we obtain, a relevant increment, with a 68,7% of vulnerable repositories being affected by either one of the two. This data can be found in Figure 8. Still, 818 repositories contain less relevant vulnerabilities classified as `MEDIUM` or `LOW`.

As one last analysis of this subsection, we took care about the number of `POCs` for each repository. A brief reminder about POCs: they're known as `Proof Of Concept` scripts; When available and if well made, even if the goal of the authors is just for educational purposes, they can be used by low skilled attackers, increasing the score seriousness. In our study we collected all the POCs available for the CVEs present in a project. We only focused on the aggregated number of POCs without dividing them by Severity since a POC can be very harmful independently of the Severity score of the associated CVE. Figure 9 shows this analysis very clearly. The good new is that we found no available POCs for 67,3% of vulnerable repositories. Even though there are repositories (8,5%) with more than 20 POCs, this can most probably be duplicates or re-uploads of the same POC in different ways and transcriptions.

D. Study of CVEs/CWEs

This study will be about CVEs and CWEs. Analysing them will give us some important and detailed statistical knowledge about the distribution and categories of vulnerabilities affecting NodeJS projects.

Figure 10 shows the distribution of CVEs by year. As expected, the majority of findings have been discovered in 2025, while the oldest bugs date back to 2013. This result underlines the fact that there are 64.6% of repositories whose dependencies have not been updated at least since 2024, ignoring Dependabot warnings or not implementing it at all. To achieve these results, we counted all CVEs among repositories, we removed duplicates, and we extracted the year from the unique CVE strings thanks to their format:

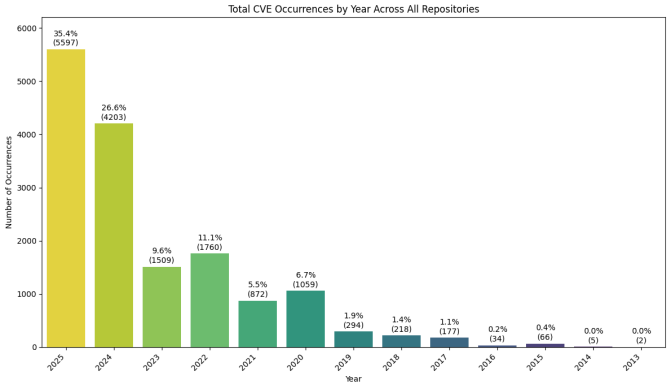

Fig. 10.

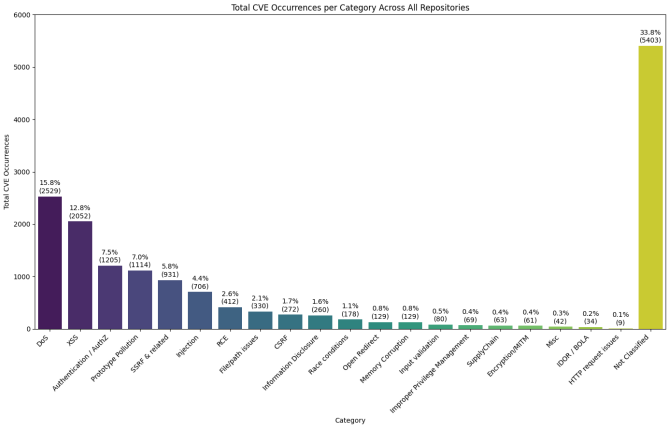

Fig. 13.

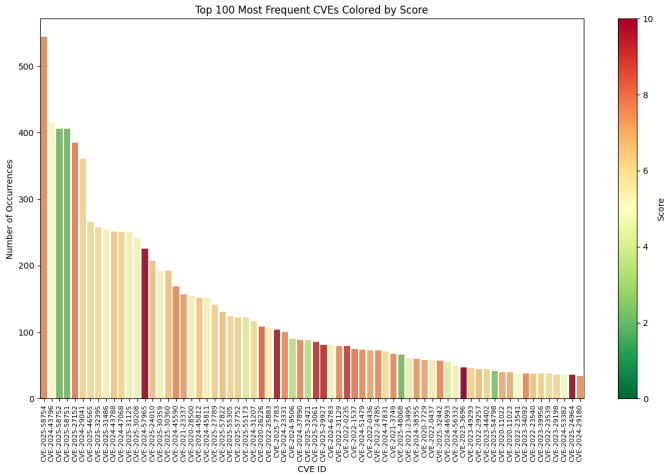

Fig. 11.

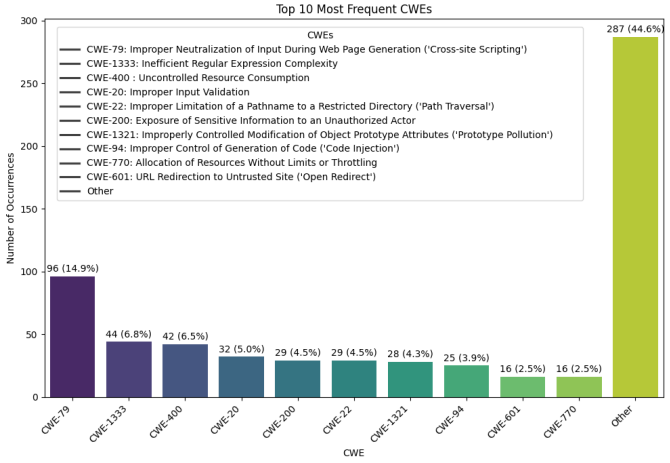

Fig. 12.

CVE-XXXX-YYYYY, where *XXXX* represents the year of disclosure.

The next analysis focused on the most frequent vulnerabilities. We show in Figure 11 the 100 most frequent CVEs. The color represents the CVSS score; red means a more critical vulnerability while green represents a less critical one. We can see that the most frequent, CVE-2025-58754, has been found more than 500 times inside different projects. This shows the importance of security in Supply Chains since a single vulnerability spread across multiple projects. The CVSS score of that CVE is 7.5; it is not a critical vulnerability but still can lead to problems to infected software. What is impressive is the solid red CVE-2024-57965 affects more than 200 projects and has a dangerous CVSS of 9.8 placing this vulnerability in the Critical category. To sum up, this figure describes how a vulnerability in one package can affect many projects further accentuating the dangers and risks of Supply Chain Attacks.

Figure 12 shows a classification of vulnerabilities by CWEs. The legend within the picture shows the common name identified by the type of vulnerability. Picking CWE-79 14,9%, we can see there is a little difference with XSS the graph shown in 13. This is because CWEs are more general definitions than vulnerability types. In our picked example, as defined by OWASP [15]:

> An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.

[15]https://owasp.org/www-community/attacks/xss/

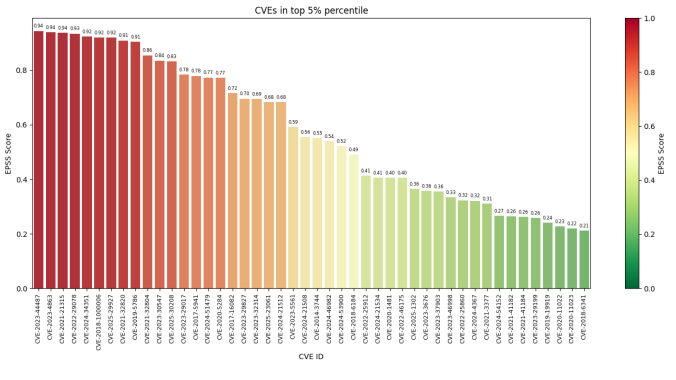

Fig. 14.

On the other hand, the CWE refers not only to injection, which can lead to a security problem for a user, but is more generic and involves the ability to inject characters that may be harmless from the security point of view, but can lead to other bugs when rendering the web page.

Finally, we have classified the unique `CVEs` found within repositories by category as already mentioned and as shown in Figure 13. As a result, we found that the most frequently categorized vulnerabilities are `DOS` (Denial of Service) and `XSS`. This is expected, especially for `XSS` as these are very frequent in a language like JavaScript that is used a lot in client-side web content frameworks. Also, `DOS` are very frequent too, most probably because of the usage of `test` built-in function. Differently from other languages, regex optimizations are not applied by this function and if not correctly sanitized, provided inputs against a particular type of regex can lead to resources consumption and eventually to a system crash [16]. Unfortunately, we were unable to classify 33.8% of CVEs, labeled `Not Classified` on the picture, because in order to obtain a classification, we asked the MITRE APIs titles and descriptions about each CVE, and a lot of them were not classifiable by reading them because they did not contain any interesting keywords for the identification. Actually, by reading some of them, we find it particularly difficult to be classified in a specific set of vulnerabilities, even if read by a human.

E. EPSS probability score

In our last analysis, we will focus on understanding the EPSS system. An EPSS score is a probability-based score that estimates the likelihood of a specific software vulnerability being exploited *in the wild* within the next 30 days, expressed as a value between 0 and 1 (or 0% to 100%). This value is not more important than the CVSS scores. As reported by first.org [17], EPSS should never be treated as a risk score because it does not account for other variables, such as the accessibility of vulnerable assets to attackers, the type of weakness the vulnerability presents, the purpose and value of the asset, etc. For example, the third to last CVE `CVE-2019-19919` [18] shown in 14, presents an EPSS score of 0.24%; however, the risk score of this vulnerability is very close to 10 (it is a critical 9.8). In this case, it is pretty obvious why the EPSS score is so low; it is a vulnerability of 2019 and the probability of being exploitable after 6 years is lower. The same reasoning applies when we have CVEs with a lower risk score but are more recent, and the probability of being exploited in the next days is much higher. Of course, the vulnerability release year is not the only parameter that influences the outcome of the EPSS score, as the formula involves more complex parameters and data to take into account as explained here: https://www.first.org/epss/model

We want to understand and show the vulnerabilities we found in the repositories and how they can affect the security of the systems. Figure 14 shows the CVEs that fall in the Top 5% percentile in the EPSS system. The colors represent the EPSS score, where red means that the vulnerability is most likely to be exploited and cause real damage. As we can see there are many CVEs that have an EPSS score higher than *0.8*. This is a real concern since they can be exploited very easily and in the very near future. We notice also that even a score of barely *0.2* is in the Top 5%. This fact shows that the EPSS system is heavily shifted towards the bottom and even scores higher than *0.2* fall in a heavily dangerous area.

V. Role of Dependabot

In our last section we will discuss what is Dependabot and why it is extremely helpful and necessary to have installed in all the repositories.

Dependabot is a GitHub service that continuously scans project dependencies and automatically submits pull requests when newer versions or security fixes are available. It covers a broad range of ecosystems such as, JavaScript (npm/Yarn), Python (pip/Poetry), Java (Maven/Gradle), PHP (Composer) and more, giving it wide language coverage. These automated PRs handle routine version bumps and security patches, vastly reducing manual upkeep. In fact, GitHub reports that in 2022 Dependabot automatically generated over 75 million pull requests. This scale shows how Dependabot has become a dominant mechanism for applying library updates and security fixes: instead of individually tracking dozens of packages, developers get actionable PRs flagged by Dependabot whenever a security patch or new release is published.

Now, we want to focus our attention on the second of the two Dependabot objectives, advising security patches for vulnerable packages. We believe that this is what makes Dependabot really essential for securing open source projects. Dependabot is a crucial tool that can mitigate the risks of

[16]https://learn.snyk.io/lesson/redos/?ecosystem=javascript#step-6c2ff686-e0a9-5296-36ba-58e8f3ce18bd

[17]https://www.first.org/epss/user-guide

[18]https://nvd.nist.gov/vuln/detail/CVE-2019-19919

supply chain attacks. Since it can be freely activated on public repositories we advocate in this direction. Even if the developer does not expect his project to be used by others, this scenario could still happen since we are talking of public code; in this situation we push for a mandatory activation of Dependabot inside every eligible project.

To sum up, Dependabot does a fantastic job in bringing to the surface obsolete and vulnerable packages inside repositories and its activation should be the first thing everyone does when creating a new repository. This simple action is one of the key steps to defend our products from catastrophic supply chain attacks.

VI. Conclusion

In this work we conducted a large-scale empirical study on the security posture of open-source JavaScript projects within the npm ecosystem. By analyzing more than 4400 repositories, we found that nearly 60% include at least one vulnerable dependency, with a worrying concentration of *critical* flaws in highly popular repositories. This confirms that the software supply chain remains an attractive attack vector, where a single compromised package can propagate to thousands of downstream projects. Our results further highlight that many vulnerabilities are neither marginal nor theoretical: 27.4% of vulnerable repositories contain at least one critical CVE, and more than two thirds include high-severity flaws, often with publicly available exploits.

A second major finding is the underutilization of GitHub Dependabot, which is capable of automatically mitigating a large fraction of these risks by keeping dependencies updated. Despite its availability, our evidence suggests that developers frequently ignore its recommendations or fail to enable it at all. This exposes a systemic gap between the existence of technical countermeasures and their adoption in practice.

From a broader perspective, our study demonstrates that supply chain security is not merely a problem of isolated vulnerabilities, but rather a structural weakness in how open-source ecosystems evolve. Addressing this challenge requires both technological enablers (e.g., default activation of tools such as Dependabot, stronger vulnerability disclosure pipelines, automated patch propagation) and cultural changes in the developer community to treat dependency hygiene as a first-class security practice.

A deeper integration of automated tools with package registries and continuous integration pipelines could significantly reduce the window of exposure to emerging vulnerabilities. Ultimately, reinforcing trust in the open-source supply chain is essential not only for protecting individual projects, but also for safeguarding the resilience of the global software infrastructure.

References

[1] Heinbockel, William J., Ellen R. Laderman, and Gloria J. Serrao. "Supply chain attacks and resiliency mitigations." The MITRE Corporation (2017): 1-30

[2] Martínez, Jeferson, and Javier M. Durán. "Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study." International Journal of Safety and Security Engineering 11.5 (2021): 537-545.

[3] "Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk", Asaf Henig and Cameron Hyde, https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/

[4] "npm Supply Chain Attack via Open Source maintainer compromise", Brian Clark, https://snyk.io/blog/npm-supply-chain-attack-via-open-source-maintainer-compromise/