2018 IEEE International Conference on Software Maintenance and Evolution

Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages

Rodrigo Elizalde Zapata*, Raula Gaikovina Kula*, Bodin Chinthanet*, Takashi Ishio*, Kenichi Matsumoto*, Akinori Ihara+
*Nara Institute of Science and Technology, Nara, Japan
+Wakayama University, Wakayama, Japan
Email: {rodrigo.elizalde.qy5, raula-k, bodin.chinthanet.ay1, ishio, matumoto}@is.naist.jp, ihara@sys.wakayama-u.ac.jp

***Abstract*—It has become common practice for software projects to adopt third-party libraries, allowing developers full access to functions that otherwise will take time and effort to create. Regardless of the migration effort involved, developers are encouraged to maintain and update any outdated dependency, so as to remain safe from potential threats including vulnerabilities. Through a manual inspection of a total of 60 client projects from three cases of high severity vulnerabilities, we investigate whether or not clients are really safe from these threats. Surprisingly, our early results show evidence that up to 73.3% of outdated clients were actually safe from the threat. This is the first work to confirm that analysis, at library level, is indeed an overestimation. The result paves the path for future studies to empirically investigate and validate this phenomena, and hopes to aid a smoother library migration for client developers.**

I. Introduction

Raising the awareness of developers to quickly update their third-party dependent library components (i.e., package dependencies) is now regarded seen as priority by both research and industry [1]–[4]. Once a newer version of a dependency is available, developers of the *client project* (i.e., a project or package that uses the dependency) are strongly recommended to update their dependency. As well as fixing bugs and adding new features, migration to a new version (i.e., updates) sometimes include fixes to prevent threats of unwanted access and potential malicious intent. Such threats are regarded as a *vulnerable dependency*. To spread the awareness, vulnerable dependencies are typically assigned a Common Weaknesses and Exposures (CWE)[1], or Common Vulnerabilities and Exposures (CVE)[2], then archived[3].

Recently, researchers have empirically shown that client developers are not practicing library migrations [5], [6], highlighting the perils and effect such as technical lag on the project, with rippling effect to the ecosystem [7]. Furthermore, studies have also shown that client developers struggle keeping up with updates, stating that either they were unaware of the opportunity, or that the cost benefit to update was a demotivating factor [1].

Being one of the most used programming language, the nodeJS libraries has become one of the most used components in contemporary software development. The npmJS platform itself hosts the largest collection of user-contributed libraries, which spreads over 700,000 packages that are been downloaded by millions of users on a daily basis. Furthermore, the npm ecosystem of packages have been the target of recent studies for researchers [4], [6], [8]. However, recent events such as the npm leftpad incident [9] and epidemic vulnerabilities such as heartbleed [10] that have spread throughout an ecosystem have triggered a reaction within the industry. This is evident by a rise in tool support and security organizations such as `skyn.io`[4] and `greenkeeper.io`[5] gaining more popularity among developers. For example, it has become common for GitHub projects to display a badge, that socially shows that the project is keeping up to date with its dependencies [2], [11].

Prior empirical studies at the component level state that developers are slow to update their dependencies. These studies suggest that developers are driven by several factors, one being the amount of migration effort (i.e., modifying client code to integrate the newer version) needed to update. However, a key threat to validity is that it was performed at the component rather than at the source code level. Recent studies [12], [13] shows that, many projects that do not actually call the affected function are safe from the vulnerability. We refer to these projects as being *clean* (i.e., they do not execute the affecting code in their client applications). Conversely, we refer to *used* clients as projects that adopt and execute the vulnerability code. Similarily, Hejderup et al. studied library migrations at the function level, constructing a call graph to understand the true effect of the vulnerability. Their study concluded that analysis at this level is indeed problematic due to execution costs needed to construct these graphs.

To analyze the impact of safe clients, we explore how *clean* and *used* client projects react to vulnerability library updates. In order to do that, We performed an exploratory study at the function level and access of the library Application Programming Interface (i.e., API). We manually identified

[1]website at https://cwe.mitre.org/
[2]website at https://cve.mitre.org/
[3]Vulnerabilities can be listed in common locations such as NVD at (https://nvd.nist.gov/) or at CWE Details at (https://www.cvedetails.com/)
[4]website at https://snyk.io/
[5]website as https://greenkeeper.io/

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00067

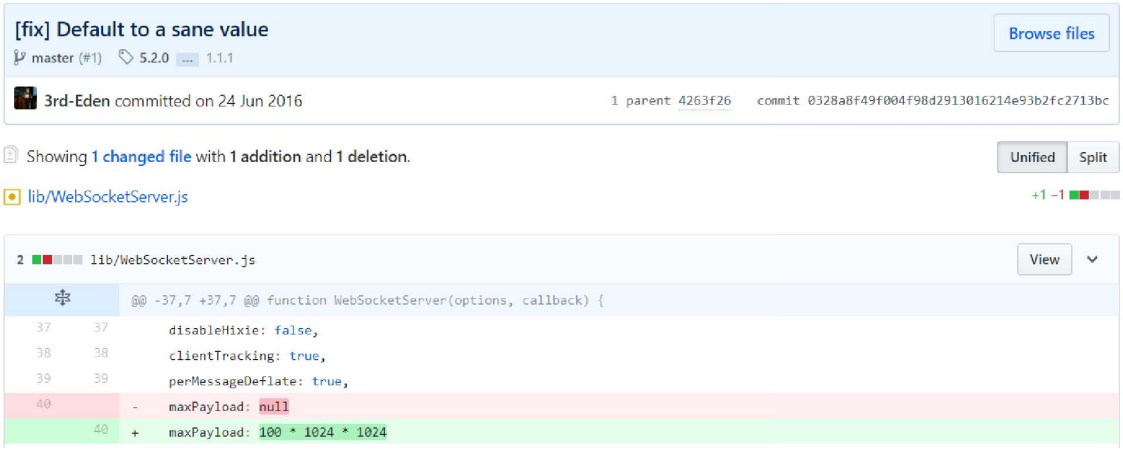

Figure 1. Vulnerability Fix that was applied to the WebSocketServer function in the ws package.

and validated vulnerability fixes and how they affected the client code. In an empirical study of npm projects and their dependencies, we manually examined a total of 60 projects, investigating three cases of high priority vulnerabilities to understand how safe and unsafe projects handle migration to safer dependency versions.

Results of the exploratory study suggest that up to 73.3% of the sampled outdated clients were indeed safe from the vulnerability threat (i.e., did not execute the vulnerable code). Furthermore, the study highlights how mapping vulnerable code to client usage is not trivial for JavaScript, with dependency information needed to understand the control flow and API detection of function difficulty. We envision that the early results of this paper will lead to further rigours studies and helps towards aiding a smoother library migration for client developers.

II. Motivating Example

WebSocket (i.e., `ws` package)[6] is described as "a simple to use, blazing fast, and thoroughly tested WebSocket client and server implementation." for the nodeJS distribution. According to the npm website, the library is very popular (with over 3,158,600 downloads and is depended upon by 3,785 other libraries) within the npm ecosystem. In 2016, the CWE disclosed a high severity vulnerability (i.e., Denial of Service (DoS)) that affected clients that used this library (i.e., https://snyk.io/vuln/npm:ws:20160624), urging clients to update if they used any `ws` versions 1.1.0 and lower. In fact, the issue was so severe, that it was also published as an advisory by the Node Security Platform (i.e., https://nodesecurity.io/advisories/120).

As shown in Figure 1, the manual inspection of the code reveals that the function `WebSocketServer` in the file `WebSocketServer.js` had been modified. Although small, it does pose as a dependency breaking issue, with client

[6]GitHub repository at https://www.npmjs.com/package/ws

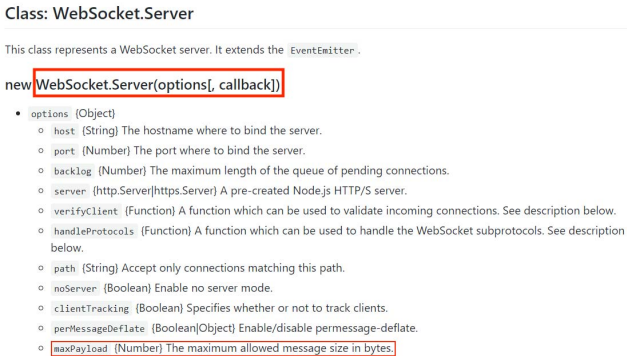

Figure 2. API documentation for the WebSocket API which is related to the WebSocketServer function.

project developer[7] stating that *'Indeed, the breaking of node.js 0.10 is precisely why we've not been able to update already'*.

After consulting the API documentation[8] and as shown in Figure 2, we find that `WebSocketServer` is not simply a function but is called from the class *WebSocket* which has the `maxPayload` as one of the option parameters.

Furthermore, due to the nature of JavaScript, there can be many different interpretations of the calling function[9]. As shown in Listing 1, we found at least three ways (i.e., (i) user-defined variable, (ii) call from the `ws` package and (iii) use the require function) that the client could call the Server function:

```
1  var wss = new WebSocket.Server({
2  ws.Server({
3  require('ws').Server;
```

Listing 1. Three ways that a client project can call the WebSocketServer function. This is through the WebSocket API.

[7]the comment is taken from the npm blog at https://github.com/node-red/node-red/issues/931

[8]website as https://github.com/websockets/ws/blob/HEAD/doc/ws.md

[9]A developer blog on the different ways to call a function highlights the variationshttps://dmitripavlutin.com/6-ways-to-declare-javascript-functions/

As a result, we find that automation detection is not a trivial task with manual inspection being required to validate the mapping between the affected function and the client-side code in JavaScript. We summarize the challenges below for JavaScript client projects:

1) *The affected function in the vulnerability fix is not the same as how it is used in the API.* We show that, how the fix affects the API, is not trivial.
2) *The exposed function call and how it is used by the client call is not the same.* Extracting the exposed functions is not a trivial task when manual validation is required.

III. Exploratory Study

The objective of the exploratory study is to understand to what extent the usage of the library vulnerable code affects the way developers update their vulnerable dependencies.

A. Approach

Figure 3 depicts an overview of our approach, which is detailed in three steps. Based on the motivating example, we used a manual investigation to validate our mapping of both (i) identification of the vulnerability fix and the affected API and (ii) identification of how the client calls the vulnerable API. We will now describe each step in detail.

- **Step One: Extract and Identify the Vulnerable Dependency and its Fix**
  From the Snyk website[10], we collected the fix information for the vulnerability issue (i.e., in the form of the pull request (PR), GitHub issue and the commit location). The output is the identification of the code fixes and the affected API that a client project may use.
- **Step Two: Identify and Collect Client Projects**
  Taking the output from Step One, for Step Two we mined and collected npmJS projects[11] from GitHub that used the vulnerable dependency. We then mined the client project repository to identify whether or not the client had migrated to a cleaner version of the dependency. This was done by inspecting the `package.json` meta-file to see whether or not the vulnerable dependency was updated to a newer version. Hence, the output is a sample of client projects that have either (i) migrated away from the vulnerable dependency (i.e., Updated) or (ii) are still dependent on the vulnerable dependency (i.e., Outdated).
- **Step Three: Usage and Update Analysis**
  Taking the outputs of Step One and Two, for Step Three we classified client projects based on (i) their update status (i.e., Updated or Outdated) and (ii) whether or not they explicitly used the affected vulnerability in their client code. As such, the output was a classification of the client projects based on the following update patterns:
  1) *Clean and Updated (CU)*: refers to clients that are using the vulnerable dependency, but were not using the affected function in their projects. These projects are deemed *safely mitigated*, as they have migrated away from the vulnerable version.
  2) *Used and Updated (UU)*: refers to clients that are using both the vulnerable dependency and the affected function in their projects. These projects are deemed *safely mitigated*, as they have migrated away from the vulnerable version.
  3) *Clean and Outdated (CO)*: refers to clients that are using the vulnerable dependency but are not using the affected function in their projects. These projects are deemed *potentially unsafe* as they have not migrated to the safe version.
  4) *Used and Outdated (UO)* : refers to clients that are using both the vulnerable dependency and the affected function in their projects. These projects are deemed *unsafe* as an attack is able to compromise the client project.

The analysis results will be reported in two sets. In the first set, we will report on the proportion of each update patterns (i.e., CU, UU, CO and UO). Our intention is to understand if using the vulnerable function has an impact on whether or not the client project dependency will be updated.

For the second set, we will analyze the time taken to update for projects that followed the CU and UU update patterns. Our intention is to understand which one of these patterns updates first.

B. Case Study Setup and Data Collection Criteria

As shown in the Table III, the three vulnerabilities were chosen due to their high severity and their impact (popularity) to the npm ecosystem of packages (i.e., we used the number of GitHub stars and downloads to rank popularity). As well as the `ws` package from the motivating example, the other two studied vulnerabilities are the popular `angular` and `marked` libraries. It is important to note that we selected security vulnerabilities that were published at least a year ago before our study, allowing ample time for client projects to become aware of the security advisories.

For the client selection, as shown in Table II, we selected the top projects based on popularity (i.e., GitHub stars, dependents and download counts). Our assumption is that popular libraries are more likely to be updated. We sampled 10 updated and 10 outdated client projects for each vulnerability, resulting in 60 client projects for the case study.

We undertook a roundtable session with four co-authors present to manually investigate each vulnerability and usage within the client. In step one, we examined the git commit history to manually trace the change commit of the vulnerability to an API. For step two, with the help of a simple regular expression search[12], we located possible locations

[10]data was mined from website https://snyk.io/vuln
[11]data was mined from the npmjs website at https://www.npmjs.com
[12]for the search we used the atom text editor to load the vulnerable version of the client and executed the search

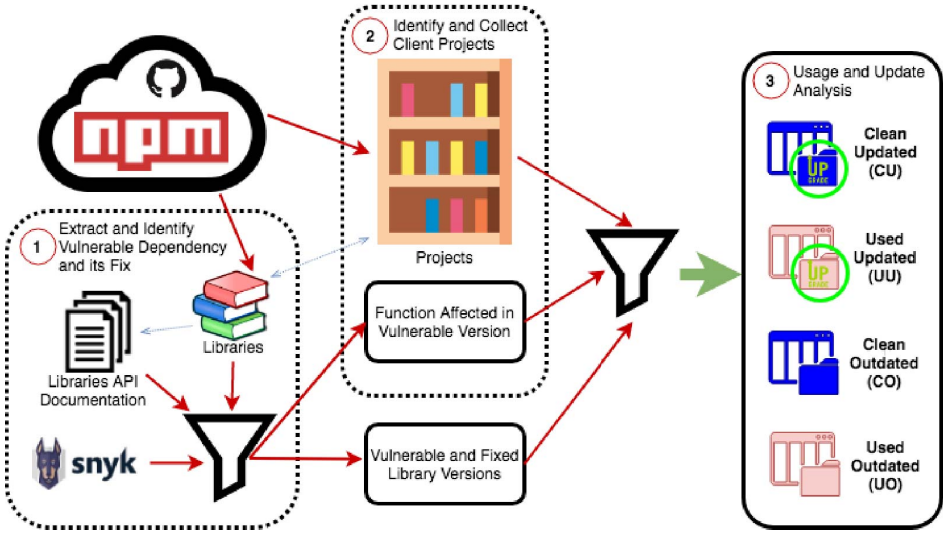

Figure 3. An Overview of our approach, comprises of three steps (i) Extract and Identify Vulnerable Dependency and its Fix (ii) Identify and Collect Client Projects (iii) Usage and Update Analysis.

Table I
SUMMARY OF THE THREE SELECTED VULNERABILITIES

Vulnerable Dependency	Severity	Snyk ID	General Description	Affected versions	Disclosed	Published in Snyk
angular	High	npm:angular:20131113	Protection Bypass	<1.2.2	12 Nov 2013	23 Jan 2017
marked	High	npm:marked:20150520	Content & Code Injection (XSS)	<0.3.6	20 May 2015	20 Apr 2016
ws	High	npm:ws:20160624	Denial of Service (DoS)	<= 1.1.0	24 Jun 2016	26 Jun 2016

Table II
SUMMARY OF THE 60 SELECTED CLIENT POPULARITY (MEASURED BY GITHUB STARS). NOTE THE 30 CLIENTS (UPDATED) HAD MIGRATED, WHILE 30 CLIENTS (OUTDATED) STILL DEPEND ON THE VULNERABLE DEPENDENCY.

Vulnerable Dependency	# GitHub stars	Updated dependents		Outdated dependents	
		Max stars	Min stars	Max stars	Min stars
angular	58,610	886	63	390	7
marked	16,459	3,614	176	11,448	1,021
ws	8,751	15,145	691	64,907	1,092

Table III
SUMMARY OF MANUAL VALIDATION MAPPING

Vulnerable Dependency	Fixed function	Mapped APIs	Keywords used in searching client code
angular	getTrustedContext	$compileProvider	$compileProvider
marked	unescape	Renderer, InlineLexer	.Renderer, InlineLexer
ws	websocketserver	websocket.server	ws, WebSocket, require('ws').Server, .Server

where the client would be using the library API. We then manually validated that the client was using the library API. Full documentation for the investigation will be available in the final version of this paper.

C. Results

(Step One): **73.3% (22 out of 30) outdated clients do not use the vulnerable code.** Figure 4 shows the results of the first step. Interestingly, while analyzing two out of the three libraries, we found that even if the client projects were using the vulnerable library most of them were not executing the affected function (UO). On the contrary, we found especially in `angular` and `marked` client projects, that a high proportion of clients that had not migrated away from the vulnerability were indeed clean of the vulnerable code. Figure 4 shows the results of how the `ws` affecting vulnerability differs from the other two vulnerabilities. One potential explanation is that the update caused a breaking change, and thus, we speculate that this would require additional migration effort.

(Step Two): **Clients that do not use the affected function code show a longer delay to migrate away from the vulnerable dependency.** Figure 5 shows, except for the clients of `ws` project, that clean updated projects (CU) had a wider spread of time taken to update. Similar to the first step, the time taken to fix the `ws` vulnerability may be related to the fact that it is a breaking change. We speculate that since the client code is not affected by the vulnerability, the developers may either be interested in keeping up to date (i.e., update as soon as possible as the migration effort will be low) or satisfied with the current state (i.e., if it is not broken, it is better not to fix).

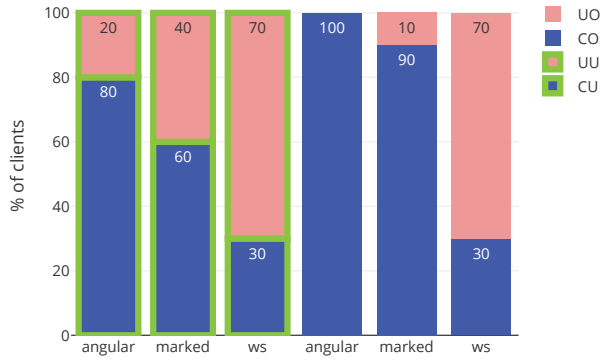

Figure 4. Proportion of each update pattern for the 30 Updated (CU, UU) and 30 Outdated (CO, UO) client projects.

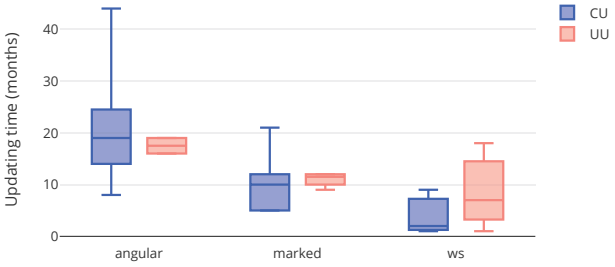

Figure 5. Time taken (months) for the updated clients (CU, UU) to migrate away from the vulnerable dependency.

IV. Implications

(1) Security vulnerability analysis at the dependency level is likely to be an overestimation. The results of the study provide evidence that many of the outdated project are free of the vulnerability. This insight is an indication that more analysis at the function level is needed to support this claim.

(2) Understanding whether or not the vulnerability affects the client code will help developers better plan their library migrations. We speculate that analysis to find whether or not the code is affecting the client code would be beneficial (especially for the novice developer) when making the decision to update.

(3) Developers should be encouraged to migrate away from the vulnerable dependency, even if the vulnerable code is not being used. As shown in the results, a significant number of clients that were clean were still found using the older version. Although there are different reasons for keeping the outdated version (i.e., fix breaks the older version or new changes are not needed), developers should be encouraged to update as soon as the fix is made available. Furthermore, we suggest that security only patches should be released. This is similar to the Debian ecosystem, where security patches are especially released and not packaged with other updates. We believe that this will help towards facilitating smoother library migrations.

(4) Automatic approaches are needed to increase the scalability of mapping the usage of library code in client projects. A potential avenue for future work is the automation of our current approach. Currently, our projects are a limited sample of the population. With automation, the same study can be conducted at a larger scale providing a more accurate comprehensive analysis.

(5) This case study should be expanded to other programming languages to generalize our results. For this case study we are using client projects that are also npm libraries. For a more rigorous study, we would like to investigate this phenomena in more generic JavaScript client projects (i.e., such as websites and frameworks written in a combination of different languages).

V. Conclusion and Future Work

This paper investigates how developers react to vulnerable dependencies based on whether or not they use the affected code in their client projects. In our study of popular npm libraries and their clients, early results suggest that up to 73.3% of the outdated clients that depend on the vulnerable dependency were in fact safe from its threat since they were not using the affected function. Immediate future work is a more rigorous replication of this study at a larger scale to validate and strengthen results.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers 18H04094, JP15H02683, and 17H00731.

References

[1] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proc. FSE*, 2016, pp. 109–120.

[2] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *Proc. ASE*, 2017, pp. 84–94.

[3] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Emp. Softw. Engg.*, vol. 23, no. 1, pp. 384–417, Feb. 2018.

[4] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proc. MSR*, 2017, pp. 102–112.

[5] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *Proc. SANER*, 2015, pp. 520–524.

[6] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proc. MSR*, 2018.

[7] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is," ser. OSS 2017, 2017, vol. 496, pp. 182–192.

[8] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proc. MSR*, 2016, pp. 351–361.

[9] "The npm blog kik, left-pad, and npm," https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm, 2018, (Accessed on 06/16/2018).

[10] "Heartbleed bug," http://heartbleed.com/, 2017, (Accessed on 06/16/2018).

[11] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding Sparkle to Social Coding : An Empirical Study of Repository Badges in the npm Ecosystem," in *Proc. ICSE*, 2018.

[12] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," in *Proc. ICSE (NIER)*, 2018, pp. 101–104.

[13] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *Proc. ICSME*, 2018.