How Do Developers Follow Security-Relevant Best Practices When Using NPM Packages?

Md Mahir Asef Kabir* Ying Wang† Danfeng(Daphne) Yao* Na Meng*
*Department of Computer Science** *Software College*†
*Virginia Tech** *Northeastern University*†
Blacksburg, USA* Shenyang, China†
{mdmahirasefk,danfeng,nm8247}.edu* wangying@swc.neu.edu.cn†

***Abstract*—Node.js has become popular among developers, partially because of its large software ecosystem of NPM (Node Package Manager) packages. When building JavaScript (JS) applications on top of NPM packages, developers can reuse the provided functionalities to improve programmer productivity. However, many NPM packages have been recently found vulnerable or malicious. Such packages can introduce vulnerabilities into their client JS applications, and realize software supply chain attacks. To reduce the impact of potentially malicious NPM packages in Node.js software ecosystem, experts suggested best practices to developers when they maintain package dependencies. These best practices include using specific commands and/or tools to (a) conduct security audit for dependencies and remove vulnerable dependencies, (b) remove unused and duplicated dependencies, and (c) fixate the version information of library/package dependencies.**

We were curious how developers followed and will follow those best practices. For this paper, we did a large-scale empirical study on 841 popularly used open-source JS applications. By analyzing their configuration files (e.g., `package.json` and `package-lock.json`), we revealed that only 32% of the applications lock the version numbers of package dependencies. The commands/tools reported (i) vulnerable, (ii) unused, and (iii) duplicated dependencies separately in 55%, 90%, and 83% of applications, which fact implies that developers often ignored the best practices we examined. We did a user study with developers to acquire their opinions on the suggested best practices and got interesting feedback. Our research will enlighten future research on the management of NPM package dependencies.

***Index Terms*—Empirical, NPM packages, best practices**

I. Introduction

Node.js is an open-source, cross-platform, JavaScript (JS) runtime environment that executes JS code outside of a web browser [1]. **NPM** (Node Package Manager) is the default package manager of Node.js. NPM consists of a command-line client, also called *npm*, and an online database of public and private packages called the *npm registry*. The npm registry hosts more than one million NPM packages, and thus provides the largest ecosystem of open-source libraries in the world [1]. Due to the widespread use of Node.js, more and more developers download NPM packages into their local software environments and develop JS applications on top of that. Developers publish and reuse NPM packages to increase programmer productivity and improve software quality.

This work was supported by US NSF Grants (NSF-1845446, NSF-1929701), US ONR Grant N00014-22-1-2057, and National Natural Science Foundation of China (Grant Nos. 62141210, 61902056).

Security experts recently found malicious or vulnerable NPM packages that can enable **software supply chain attacks**—the attacks that hackers achieve by manipulating code or exploiting vulnerabilities in third-party software libraries, to compromise the **client applications** that use those libraries [2]. For instance, In March 2021, security experts identified a vulnerability in the NPM package `netmask`, which could expose private networks and lead to a variety of attacks such as malware delivery [3]. As `netmask` is highly popular, the vulnerability can affect more than 278,000 client projects once it is exploited. The usage of NPM packages can expose client applications to security risks. Therefore, to reduce risks, domain experts usually recommend application developers to follow best practices when maintaining package dependencies [4]–[8]. After extensively searching online with keywords "npm best practices", we found the following three best practices (BPs) most popularly suggested:

BP1: Scan vulnerabilities in library dependencies using command "`npm audit`" and remove vulnerabilities with "`npm audit fix`" [4]–[11], because eliminating the usage of vulnerable packages can reduce the malicious module attack surface. The command "`npm audit`" submits to *npm registry* a description of dependencies in a JS application, and asks for a report of known vulnerabilities in those packages. If any vulnerability is found, the impact and appropriate remediation will be calculated. If the argument `fix` is also provided, i.e., "**`npm audit fix`**", then remediation will be automatically applied to the dependency tree.

BP2: Scan and/or remove unused and duplicated packages using "`depcheck`" and "`npm dedupe`" [4], [5], [7], because these dependencies can unnecessarily grow the attack surface and add clutter to software implementation. "**`Depcheck`**" [12] is a tool to analyze the dependencies in a project, and to **identify any unused** direct or indirect (libraries direct dependencies depend on) NPM dependency. "**`Npm dedupe`**" [13] is a command to **reduce duplication** in a dependency tree. It searches the tree for any package (regardless of the version numbers) that is depended on by multiple packages, and tries to simplify the overall structure by moving packages up the tree and merging redundant dependencies.

BP3: Enforce the lock file `package-lock.json` to pin library dependency versions [5], [6], [8], [10], so that developers can obtain deterministic installations of packages

across different environments. `Package.json` is a JSON file that exists at the root of a JS/Node project. It holds metadata relevant to the project and it is used for managing the project's dependencies, scripts, version, etc. [14]. When developers specify version ranges for package dependencies in `package.json` (e.g., "˜5.2.0" means the value range "$\geq$5.2.0 &<5.3.0"), *npm* has the freedom to decide which version to download within those ranges. Such freedom can introduce nondeterministic package downloads as well as installations across different environments [6], and pull in vulnerable or malicious package versions. For deterministic installations, JS developers are recommended to automatically generate lock files `package-lock.json` that explicitly document the exact version info, and **commit lock files to software repositories**.

With these best practices suggested for years, we were curious about the following research questions:

- **RQ1**: How well did developers follow best practices?
- **RQ2**: How well can existing tools address developers' violations of best practices?
- **RQ3**: In the scenarios when developers do not follow best practices, what are the reasons?

To answer these questions, we conducted a large-scale empirical study on 841 popularly used open-source JS applications. After cloning the repositories on GitHub, we studied each application by taking three steps. First, to identify developers' violations of any of the best practices mentioned above, we analyzed the configuration file(s) (e.g., `package-lock.json`) and JS code by applying our own scripts, existing *npm* commands, and related tools. Second, for the detected violations, we applied existing tools to address issues and to explore the feasibility for developers to follow best practices. Third, we sampled 60 detected violations and filed pull requests (PR) to examine developers' willingness to follow the best practices.

Our study revealed interesting phenomena. Only 32% of the studied applications specify the exact versions of all library dependencies, most of which started using `package-lock.json` file in 2017–2018 to lock dependencies. Most applications (e.g., 65%) violate the best practice of fixating dependency versions, as they allow distinct versions of libraries to flexibly match the version-range specification. The scripts, commands, and/or tools we used reported vulnerable, unused, and duplicated dependencies separately in 460, 755, and 698 applications. These numbers imply that most developers did not make every effort to follow the suggested best practices. Although there is sufficient tool support to generate `package-lock.json` files, we do not have enough tools to remove unneeded and vulnerable packages. Developers expressed two major reasons to explain their decision-making. First, developers did not see the necessity of following best practices. Second, developers found the outputs of recommended tools to be wrong.

Our scripts and datasets are available at https://figshare.com/s/c7e6647a6d5b4cc8a2d7

II. Methodology

To understand how developers follow the best practices concerning dependency security, we first created a large dataset of JS applications, and took three steps to investigate our research questions mentioned in Section I.

Dataset Creation: We identified subject programs by referring to the top 1,000 most depended-upon packages [15]. We started with this initial set of JS applications for two reasons. First, each of the applications has a package published at *npm registry*. It means that the applications are likely to have `package.json` files to be checked for our investigation, and *npm* commands are probably executable with them. Second, the most depended-upon packages are definitely popularly used JS applications. Analyzing these subject applications can enhance the representativeness of our empirical results. In the list of 1,000 JS applications, we successfully located the GitHub repositories for 980 applications, and 919 of them have `package.json` in the most recent versions. By removing duplicated repositories, we got a dataset of 841 repositories.

Step 1: Investigating how well developers followed best practices (RQ1): For best practice (a), we applied "`npm audit`" (i.e., npm-audit) to check whether any project depends on known vulnerable NPM package(s). For best practice (b), we applied "`depcheck`" and "`npm ls --all`" to separately reveal unused and duplicated dependencies. In particular, "`npm ls --all`" prints a dependency tree for each JS application. We wrote a script to scan every tree, and to decide whether any package occurs multiple times in a tree; if so, the package is duplicated. For best practice (c), we scanned the projects downloaded from GitHub for (1) the version spec of library dependencies in `package.json`, and (2) the existence of `package-lock.json`. If a project has no lock file and specifies version ranges for some dependencies, we consider it to violate the best practice.

Step 2: Investigating how well existing tools can address developers' violations to best practices (RQ2): For any application that violates best practice (a) by using vulnerable packages, we applied "`npm audit fix`" to tentatively address those issues. Afterwards, we reapplied "`npm audit`" to check whether all vulnerable packages were successfully replaced.

Concerning best practice (b), if an application has unused packages reported, we did not try to address the issues because there is no tool to perform that task. If an application has duplicated dependencies reported, we first applied "`npm dedupe`" to remove all duplicates. We then reapplied "`npm ls --all`" to list the dependency tree and find remaining duplicates.

Concerning best practice (c), as `package-lock.json` is automatically generated whenever *npm* modifies either `pacakge.json` or the dependency tree, we did not further explore how well this feature works; it seems that the default mechanism works well to generate lock files. Developers only need to always commit their lock files to software repositories, in order to ensure consistent build results.

Step 3: Investigating what developers think of their best practice violations (RQ3): We sampled 20 violations for each best practice, ensured the violations to be from distinct projects, and filed pull requests (PRs) to interact with different developers. In each PR, we described the recommended best practice, the security implication of that best practice, any vi-

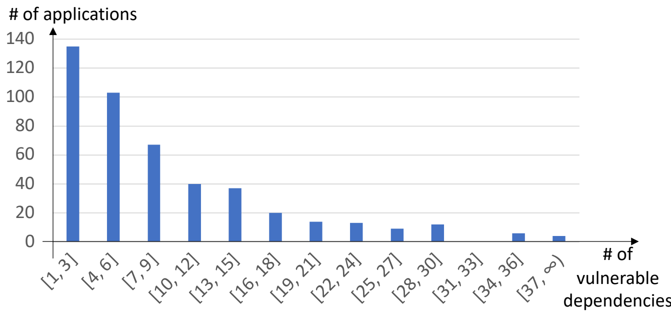

Fig. 1: The distribution of 460 JS applications based on their numbers of reported vulnerable dependencies

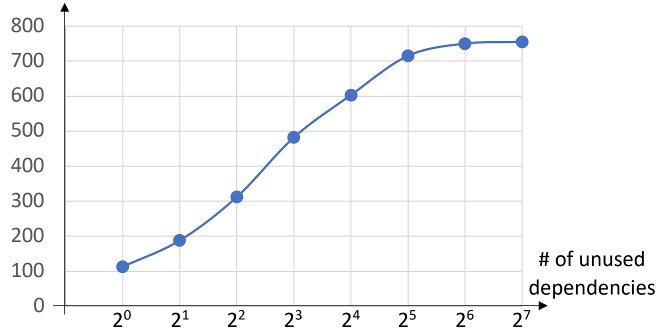

Fig. 2: The CDF of 755 JS applications based on the unused dependencies reported

olation we found in the codebase, and our suggested solution. We asked developers for their thoughts on those suggestions and their reasons for violating best practices.

III. Experiment Results

This section presents and explains our results for each RQ.

A. Developers' Compliance with Best Practices

By running "`npm audit`" with the latest software versions of 841 repositories, we found the command to report **vulnerabilities** in 460 (55%) programs, no vulnerability in 152 (18%) projects; it failed to execute in the remaining 229 ones.

Among the 229 cases where errors were reported, the command cannot identify any appropriately generated lock files, nor can we reproduce lock files for a variety of reasons (e.g., no access to private packages or unsupported URL types). The command runs smoothly with 612 projects, among which there are more applications with reported vulnerabilities than those without (460 vs. 152). For the 460 vulnerable applications, Fig. 1 shows their distribution based on the counts of detected vulnerable dependencies. Generally speaking, as the number of vulnerable dependencies increases, the number of applications decreases. As shown in the figure, most programs (i.e., 238) have 6 or fewer vulnerable dependencies; 222 programs have at least 7 vulnerable dependencies. All numbers mentioned above imply that according to "`npm audit`", vulnerable dependencies popularly exist in JS applications. It seems that developers paid little attention to vulnerable dependencies, or did little to intentionally reduce those vulnerabilities.

> **Finding 1:** *Most developers did not seem to follow best practice (a), as "npm audit" reported lots of vulnerabilities.*

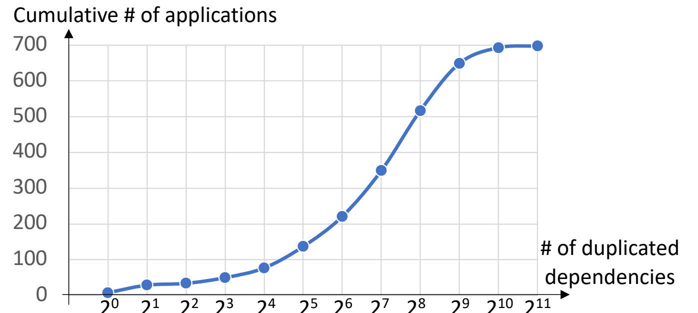

Fig. 3: The CDF of 698 applications based on the duplicated dependencies reported

Unused dependencies were reported in the latest versions of 755 programs. Fig. 2 shows the cumulative distribution function (CDF) of 755 programs based on the number of unused dependencies reported. In the figure, we use a log scale for the x-axis, as the number of unused dependencies varies a lot across applications. Each application has 1–127 unused dependencies reported. For each label 2^n on the x-axis, the y-value counts the applications that have at least 1, and at most 2^n unused dependencies. According to the figure, 715 programs have $[2^0, 2^5]$ or 1–32 unused dependencies. The median count of unused dependencies per project is 6, while the mean value is 10. These numbers imply that many programs have one or multiple unused dependencies reported.

> **Finding 2:** *755 out of 841 have unused dependencies reported by "depcheck" in their latest versions.*

Duplicated dependencies were reported in the most recent versions of 698 programs. Fig. 3 shows the CDF of 698 programs based on the duplicated dependencies reported. According to Fig. 3, the majority of programs have 17-512 duplicates in their separate dependency trees. The median count of duplicated dependencies is 129, and the mean is 192.

> **Finding 3:** *698 out of the 841 projects have duplicated dependencies reported. Most projects seem to have alarmingly large numbers of duplicates reported (i.e., 17–512).*

In terms of **lock files**, we observed 269 programs to have `package-lock.json` files in version history. There are 21 programs that have no dependency on any NPM package, so they do not need any lock file to fixate any library dependency. Another three programs specify the exact version numbers of all package dependencies in `package.json` files, so they do not need any lock file either. Namely, in only 35% (293/841) of the studied cases, we have no difficulty reproducing the original software environment where developers build and execute their programs. The other 548 programs have no lock file in GitHub repositories, although they all have library dependencies.

Since 2017 when NPM 5 was released, `package-lock.json` has been automatically generated for any *npm* operation that modifies the `node_modules` tree or `package.json`. The lock file describes the exact tree that was generated in the developers' environment, so that subsequent installs can generate identical trees regardless of intermediate dependency updates [16].

It is unknown how soon developers adopted the best practice

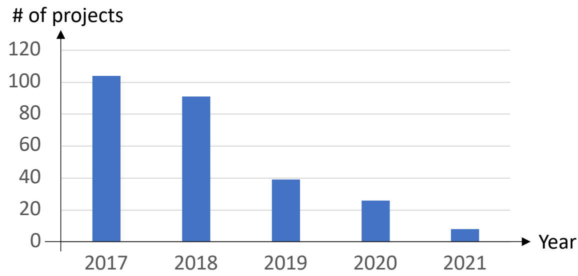

Fig. 4: The distribution of 268 projects based on the year when lock files were initially introduced to repositories

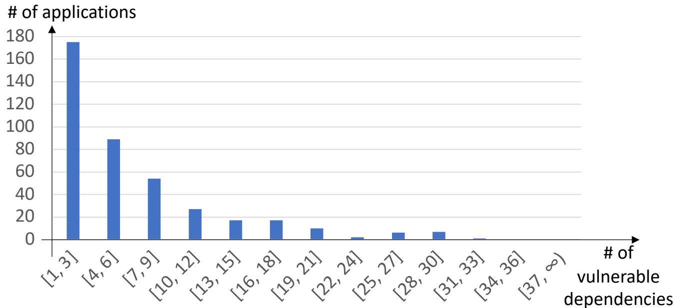

Fig. 5: The distribution of 405 fixed applications based on their numbers of vulnerable dependencies

of committing lock files to version history, so we mined the version history of 841 projects. We found that 822 of the projects were created in or before 2017. Among these projects, 268 projects have lock files in their latest versions. We further located the commits that initially introduced lock files into the 268 repositories. As shown in Fig. 4, 104 programs added lock files to repositories in 2017; this number slightly went down to 91 in 2018, and dropped significantly to 39 in 2019; in 2021, only 8 programs added lock files. The figure implies that the developers who care about deterministic installs typically committed lock files to repositories as early as possible. If some developers did not introduce lock files into repositories earlier, they may also be reluctant to add those files later.

Finding 4: *548 projects have no lock file under version control. Among the 268 projects we further studied, 195 projects added `package-lock.json` files in 2017 or 2018.*

To sum up, our results show that most developers did not follow best practices (a)–(c). Although a good number of programs (i.e., 269) have lock files added to GitHub repositories, we have not seen the best practice widely spread among projects over years.

B. Current Tools to Remove Violations of Best Practices

For the 460 applications with vulnerable dependencies reported in the latest software versions, we applied "`npm audit fix`" to address issues, and to replace vulnerable versions with the secure ones as *npm* suggests. Unfortunately, we found only 55 applications to have the issues fully resolved; 164 applications have issues partially removed; and 241 applications have issues remain. In other words, 405 (i.e., 164 + 241) programs still have vulnerable dependencies. Fig. 5 shows the distribution of 405 fixed programs based on the counts of vulnerable dependencies. By comparing Fig. 5 with Fig. 1, we saw that

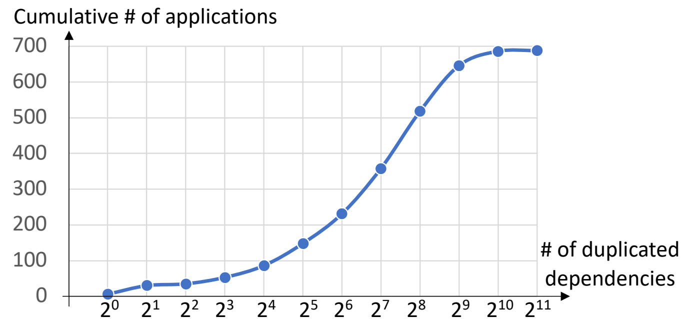

Fig. 6: The distribution of 688 revised applications based on their counts of duplicated dependencies reported

the fixed programs have a lot fewer vulnerable dependencies. 264 fixed programs have 1–6 vulnerable dependencies, and no fixed program has a count beyond 33.

Our observations imply that "`npm audit fix`" can reduce vulnerable dependencies for some applications; however, it still cannot remove all vulnerable dependencies for most programs. One possible reason can explain such deficiency: the secure alternatives of some vulnerable versions do not satisfy the version-range spec in `package.json`. For instance, package `hapi` has versions 6.1.0 and earlier known to be vulnerable to a rosetta-flash attack, which can be used by attackers to send data across domains and to break the browser same-origin-policy [17]. To remedy any program depending on the vulnerable versions, *npm* needs to replace those versions with versions $\geq$6.1.1. However, if a program specifies "1.x" for `hapi` in `package.json`, npm-audit does not replace the vulnerable version because the secure alternative 6.1.1 or later does not satisfy that version spec. More advanced tools or fixing strategies are still needed to eliminate vulnerable dependencies in a larger portion of JS applications.

Finding 5: *The command "`npm audit fix`" removed all vulnerable dependencies in only 55 programs, but kept vulnerable dependencies as they were in 241 programs.*

For the 698 applications with duplicated dependencies reported in the latest version, we applied "`npm dedupe`" (i.e., npm-dedupe) to eliminate duplicates. Unfortunately, we found only 10 applications to have duplicates fully removed; 467 applications have duplicates partially removed; 113 applications have duplicates unchanged; 118 applications have duplicates increased. Namely, 688 of the 698 applications still have duplicates reported. The median count of duplicates is 119, and the mean value is 179. Fig. 6 shows the distribution of 688 applications based on the counts of reported duplicates. When comparing Fig. 6 with Fig. 3, we found the figures to be almost the same.

Our observations imply that although npm-dedupe can provide some help to reduce duplicates in certain circumstances, it does not always work effectively. Instead, it kept duplicates unchanged or even increased duplicates in many applications. We manually inspected the dependency trees of some packages, but could not identify any characteristics to explain the command's ineffectiveness. The command may have implementation issues.

TABLE I: Developers' feedback on our PRs

Category	Feedback	# of PRs
Audit	Developers considered the vulnerabilities to be false positives.	4
	Developers accepted the suggested fix	3
Unused	Developers believed the reports to be false positives.	4
	Developers partially agreed, and would like to remove some reported unused packages.	1
	Developers did not worry about unused dependencies.	2
Duplicated	Developers explained the necessity to have duplicated dependencies.	1
	Developers agreed that duplicated dependencies should be removed, but they did not trust npm-dedupe.	1
	Developers assumed the package manager can reduce duplication by default.	1
	Developers did not worry about duplicated dependencies.	1
Lock	Developers did not see the need to pin dependency versions.	4

Finding 6: *Npm-dedupe removed all reported duplicates in only 10 programs, partially removed duplicates in 467 programs, but kept or increased duplicates in 231 programs.*

C. Reasons for Violations of Best Practices

After filing 60 pull requests (PRs) based on the detected violations of best practices, we received 22 responses. As summarized in Table I, only 4 out of 22 developers are (partially) positive about our PRs; they either modified or will modify projects accordingly.

1) Feedback for audit-related PRs: Among the seven PRs related to npm-audit, two PRs received the same response from distinct developers—a reference to article "*npm audit: Broken by Design*" [18]. The article complained that "*in many situations, (npm-audit) leads to a 99%+ false positive rate, creates an incredibly confusing first programming experience, ..., and at some point will lead to actually bad vulnerabilities slipping in unnoticed.*" As npm-audit offers no concrete security exploit to demonstrate any successful attack, developers wonder whether the exploits are only achievable when hackers can access their machines. Consequently, developers lose trust in vulnerability reports, because if hackers can control developers' machines, they can cause more severe problems than just hacking a JS application.

Another two PRs are about either a vulnerable package listed in the "`devDependencies`" segment inside `package.json`, or a vulnerable dependency of the test suite. Developers believed the issues to be irrelevant, as the vulnerable dependencies will not get included into the released software products. All the above-mentioned concerns on npm-audit make sense to us. They imply that developers care about security and vulnerable dependencies, but they are unsatisfied with npm-audit and desperately need better tools. Three of our PRs were accepted by developers. They responded with "*Audit fixes were applied, thanks*" or "*Working on this*".

Finding 7: *Developers cared about security. However, some developers felt npm-audit to be broken when reporting vulnerabilities, as the tool has many false positives.*

2) Feedback on PRs related to unused or duplicated dependencies: Among the seven PRs related to unused dependencies, four PRs were considered false positives. One developer explained "*The detection is wrong. For instance, check-dts is used in package scripts ...*" As shown in Listing 1, `check-dts` is mentioned in the test script of the project's `package.json`, but "`depcheck`" incorrectly reported it unused. For a fifth PR, the developer considered the report partially correct: some used packages are wrongly diagnosed as unused. However, the developer would like to remove the ones correctly reported. For the remaining two PRs, developers believed that the unused dependencies are irrelevant to core files and cause no runtime issue, so they decided to change nothing. All developers' responses are meaningful and valuable to us, as they all indicate limitations of the recommended tool "`depcheck`".

Listing 1: A `package.json` file that uses check-dts

```
...
"scripts": {
  "test": "jest --coverage && eslint . bin/* && check-dts
       && size-limit"
} ...
```

Finding 8: *Most developers considered the reported unused dependencies as (1) false alarms or (2) unimportant issues. One developer wanted to fix the correctly reported issues.*

Four PRs are related to duplicated dependencies. For one PR, developers clarified their intent of having duplicated dependencies. Currently, npm-dedupe considers distinct versions of the same package within a dependency tree to be duplicates. However, developers preferred treating those multiple versions as distinct dependencies to preserve the differences between versions. For another PR, developers agreed that duplicated dependencies should be removed, but they did not trust npm-dedupe to automatically reorganize dependency trees. For a third PR, developers believed that the duplicated dependencies were introduced by the tree produced by "`devDependencies`". As those dependencies will not get included into the released software, developers felt no obligation to remove the reported duplicates. We agreed on all concerns mentioned above.

For a fourth PR, developers believed that by default, `npm` could reduce duplicates when installing dependencies, so they did not see the need to specially invoke "`npm dedupe`". Based on our experience, nevertheless, the default `npm` installer often does not reduce duplicates effectively. Specially invoking "`npm dedupe`" helps remove more duplicates.

Finding 9: *Most developers did not worry about duplicated dependencies. Meanwhile, some developers expressed (1) the necessity of keeping multiple versions of the same package, and (2) their concern on depcheck's reliability.*

3) Feedback for lock-related PRs: For all four PRs, developers considered it unnecessary to lock dependency versions. A developer mentioned "*... packages should not have lockfiles. Consumers should always have a lockfile in their application, which fully addresses all the concerns described ...*". This explanation seems contradictory to us. When a package P depends on other packages, the package itself is a consumer of packages and should have a `package-lock.json` file included into its software repository. Unfortunately, the developers we contacted did not consider their own projects to be consumers

of other packages. Another developer believes it OK to omit lock files, as all packages on which his/her project depends will always follow semantic versioning. However, this belief does not always hold as people observed incorrect semantic versions to cause issues when lock files are missing [19], [20].

Finding 10: *Developers did not bother to use lock files, mainly because they did not care about reproducible builds or did not fully understand the locking mechanism.*

IV. Our Recommendations

Our work presented significant gaps between the recommended best practices of NPM dependency maintenance and developers' actual practices, as well as between the expected tool support and existing tools' capabilities. Such gaps may result in serious software vulnerabilities and cause various issues to end users. Below are our recommendations:

a) ***For Developers**:* Generate lock files to fixate the package dependencies for any released version of their JS projects, and put those files under version control. Without lock files, the npm client may install dependencies into the `node_modules` directory nondeterministically. This means that due to the installation order and time of dependencies, the structure of a `node_module` directory can be different from one person to another. These differences can cause hard-to-reproduce bugs that take a long time to fix [20], [21].

b) ***For Tool Builders**:* Improve existing tools or create new tools to better meet developers' needs. Concerning security audit, developers consider npm-audit to generate too many false positives, as the tool does not provide any evidence to show how each reported vulnerability can be actually leveraged by hackers. Concerning the detection of unused or duplicated dependencies, developers also reported false alarms produced by existing tools. Our experiment results show that there is insufficient tool support to help developers properly address their violations of best practices. We still need more advanced tools that conduct sophisticated program analysis, to generate customized project-specific attacks, to improve detectors' accuracy, and to generate high-quality fixing suggestions.

c) ***For Researchers**:* Cautiously use existing npm commands and tools when conducting empirical studies and reporting experiment results; invent and adopt better tools to improve the rigor of empirical findings. We noticed that some recent studies were conducted based on the usage of depcheck [22], [23] and npm-audit [24], [25]. However, depcheck can falsely report unused dependencies [26] (see Section III-C2). Npm-audit reports vulnerabilities based on the existence of vulnerable package dependencies, instead of based on the finer-granularity code analysis or enabled security exploits. Developers did not seriously treat the reported vulnerabilities most of the time, as reflected by our experimental measurements (Section III-A) and user study (Section III-C3). Therefore, researchers need to be very careful when summarizing the security of NPM ecosystem by using npm-audit.

V. Threats to Validity

a) Threats to External Validity: All observations we made are limited to the selected JS projects, and the selected developers who responded to our PRs. The observations may not generalize to unchosen projects or developers who did not receive or reply our PRs. In the future, we plan to include more projects and involve more developers into our empirical study, so that our findings are more representative.

b) Threats to Internal Validity: Some tools like yarn [27] and snyk [28] can serve as alternatives to the commands/tools suggested in the best practices. We did not experiment with these additional tools due to their limited availability and less popularity. However, most of our observations are likely to preserve even if we used the alternatives. For instance, similar to npm-audit, snyk-test also reveals vulnerable dependencies based on a predefined vulnerability database. It does not customize security attacks for any JS project, so it is unlikely to better satisfy developers. As with npm, yarn can also generate lock files to fixate versions of package dependencies. Nevertheless, if developers do not see the need to pin dependency versions, yarn cannot better help developers, either.

VI. Related Work

Various research was conducted to help improve developers' secure coding practices [29]–[35]. The research most related to our paper includes the empirical studies to characterize NPM packages and their dependencies [19], [36]–[42]. Specifically, Wittern et al. analyzed the NPM ecosystem and found that package dependencies increased over time, although many projects depend on a core set of packages. Cogo et al. explored why developers downgraded package dependencies; they revealed reasons like (1) defects in a specific version of a provider, (2) unexpected feature changes in a provider, (3) incompatibilities, and (4) prevention of issues introduced by future releases [42]. Decan et al. [39] and Zerouali et al. [40] studied how soon developers updated their dependencies after the new package releases became available. Both groups reported that a lot of dependency information was updated weeks or months later than the introduction of new releases.

Compared with prior work, our study is different in terms of the research scope and study method. We summarized the best practices widely suggested by NPM-related tutorials, and examined how and why developers violated those rules.

VII. Conclusion

This study assesses how developers follow security-related practices when using NPM packages. We analyzed 841 JS repositories, and observed interesting phenomena. Developers are often recommended to use certain commands/tools to (1) scan for and remove vulnerable dependencies, (2) remove unused and duplicated packages, and (3) add lock files to software version control. However, in reality, even though tools can report violations of the above-mentioned best practices, current tools seldom fix those violations and developers rarely treat tool outputs seriously. In the future, we will better define best practices, and build tools to (i) better reveal violations of best practices as well as (ii) synthesize attacks that exploit vulnerable dependencies.

Acknowledgment

We thank anonymous reviewers for their valuable feedback.

REFERENCES

[1] "Exploring the JavaScript Ecosystem: Popular Tools, Frameworks, and Libraries," https://mirzaleka.medium.com/exploring-javascript-ecosystem-popular-tools-frameworks-libraries-7901703ec88f, 2020.

[2] "Software supply chain attacks – everything you need to know," https://portswigger.net/daily-swig/software-supply-chain-attacks-everything-you-need-to-know, 2021.

[3] "Vulnerability in 'netmask' npm Package Affects 280,000 Projects," https://www.securityweek.com/vulnerability-netmask-npm-package-affects-280000-projects, 2021.

[4] "Controlling the Node.js security risk of npm dependencies," https://blog.risingstack.com/controlling-node-js-security-risk-npm-dependencies/\#3areotherdevelopersusingthispackage, 2016.

[5] "npm package best practices," https://co-pilot.dev/npm-package, 2021.

[6] "10 npm Security Best Practices," https://snyk.io/blog/ten-npm-security-best-practices/, 2019.

[7] "NPM Tips and Tricks," https://blog.bitsrc.io/npm-tips-and-tricks-24c5e9defea6, 2020.

[8] "Npm Security Best Practices," https://bytesafe.dev/posts/npm-security-best-practices/\#no9-deterministic-results, 2021.

[9] "We're under attack! 23+ Node.js security best practices," https://medium.com/@nodepractices/were-under-attack-23-node-js-security-best-practices-e33c146cb87d, 2018.

[10] "Top 10 Npm Security Best Practices," https://dev.to/danielp/top-10-npm-security-best-practices-2lp9, 2021.

[11] "NPM security best practices – SoluteLabs," https://www.solutelabs.com/blog/npm-security-best-practices, 2020.

[12] "depcheck," https://www.npmjs.com/package/depcheck, 2021.

[13] "npm-dedupe," https://docs.npmjs.com/cli/v7/commands/npm-dedupe, 2021.

[14] "Understanding the package.json file," https://blog.ezekielekunola.com/understanding-the-package.json-file, 2020.

[15] "npm rank," https://gist.github.com/anvaka/8e8fa57c7ee1350e3491, 2020.

[16] "package-lock.json," https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json, 2021.

[17] "Rosetta-Flash JSONP Vulnerability," https://www.npmjs.com/advisories/12, 2021.

[18] "npm audit: Broken by Design," https://overreacted.io/npm-audit-broken-by-design/, 2021.

[19] P. Goswami, S. Gupta, Z. Li, N. Meng, and D. Yao, "Investigating the reproducibility of npm packages," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 677–681.

[20] "Semantic Versioning Sucks! Long Live Semantic Versioning," https://developer.okta.com/blog/2019/12/16/semantic-versioning, 2019.

[21] "Should I commit the yarn.lock file and what is it for?" https://stackoverflow.com/questions/39990017/should-i-commit-the-yarn-lock-file-and-what-is-it-for, 2016.

[22] A. Javan Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[23] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the untriviality of trivial packages: An empirical study of npm javascript packages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[24] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" Master's thesis, Delft University of Technology, 05 2015.

[25] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Smallworld with high risks: A study of security threats in the npm ecosystem," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, pp. 995–1010.

[26] "Incorrect unused dependencies," https://github.com/depcheck/depcheck/issues/440, 2019.

[27] "Yarn - Package Manager," https://yarnpkg.com, 2021.

[28] "Snyk," https://snyk.io/, 2021.

[29] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *ICSE*, 2018.

[30] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 536–547.

[31] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 2455–2472. [Online]. Available: https://doi.org/10.1145/3319535.3345659

[32] M. Islam, S. Rahaman, N. Meng, B. Hassanshahi, P. Krishnan, and D. D. Yao, "Coding practices and recommendations of spring security for enterprise applications," in *2020 IEEE Secure Development (SecDev)*, 2020, pp. 49–57.

[33] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. D. Yao, and N. Meng, "Automatic detection of java cryptographic api misuses: Are we there yet," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[34] D. Yao, S. Rahaman, Y. Xiao, S. Afrose, M. Frantz, K. Tian, N. Meng, C. Cifuentes, Y. Zhao, N. Allen, N. Keynes, B. Miller, E. Heymann, M. Kantarcioglu, and F. Shaon, "Being the developers' friend: Our experience developing a high-precision tool for secure coding," *IEEE Security & Privacy*, no. 01, pp. 2–11, 2022.

[35] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. Yao, and N. Meng, "Example-based vulnerability detection and repair in java code," in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, 2022, pp. 190–201.

[36] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 351–361.

[37] A. Trockman, S. Zhou, C. Kastner, and B. Vasilescu, "Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 511–522.

[38] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, May 2018, pp. 181–191.

[39] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2018, pp. 404–414.

[40] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse*, R. Capilla, B. Gallina, and C. Cetina, Eds. Cham: Springer International Publishing, 2018, pp. 95–110.

[41] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the diversity of software package popularity metrics: An empirical study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019, pp. 589–593.

[42] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.