A Comparison of Vulnerable Dependency Detection Tools

Abstract

The goal of this study is *to aid security practitioners and researchers in choosing the right tooling and determining research directions for investigating vulnerable dependency through a comparative study of existing security tools*.

1 Introduction

A modern software typically uses several third-party libraries, packages, or frameworks, usually referred to as *dependencies*. As much as 80% of code in today's applications can come from these packages, in a form of both direct and transitive dependency [23]. A scan over 1,100 commercial codebases by Black Duck in 2017 found that 96% of the applications contain open source packages with an average 257 packages per application, and that the average usage grew from 36% in the previous year to 57% [25]. NPM, a package manager for the JavaScript programming language, hosts more than 1.3 million open source packages at the time of this writing [1].

However, known vulnerabilities in dependencies is one of the top ten security risks [17]. In 2017, attackers exploited a vulnerability in Apache Struts Framework of Equifax software leading to a major data breach. While the vulnerability was published and patched in March of that year, attacks were successful till July as Equifax kept using the vulnerable version of the framework [12]. Since this incident, a recent security report has found a 430% surge in cyber attacks aimed at upstream open source libraries from July 2019 to May 2020 [21].

Several security tools, both open source and commercial, exist that detect the dependencies of a software with some known vulnerabilities, i.e. *vulnerable dependencies* (VD). However, like many other security tools [13, 24], developer response to VD alerts may not always result in simply *fixing* the vulnerability by removing the dependency or changing it to a safer version. A recent study that analyzes GitHub's dependabot alerts on VD, finds fix rate to stand only at 26.9% [[cite arxiv]]. Typically, false positives, unexploitability, and cost of fix are among the many reasons that security alerts may remain ignored [14, 15].

While tools exist to detect VDs for a software project, a formal study is yet to be performed to review the techniques and data sources these tools use; and how their results compare to each other. Further, different tools may rank their results based on various metrics and provide developers with additional information to aid them in prioritizing fixes. Prioritization of fixes for VDs based on risk may be necessary in the real world due to the various costs involved in each fix, including but not limited to, regression testing; possibility of breaking changes; patching all devices running the software; legacy applications, inadequate workforce [19] etc. A 2017 survey of 340 industry professionals found that 72% of the respondents fear that security updates could break production systems [3].

A comparative evaluation of existing tools will help security practitioners in choosing the right tooling for their respective use cases and help researchers to identify the cutting-edge and existing gaps to determine future research directions. The goal of this study is *to aid security practitioners and researchers in choosing the right tooling and determining research directions for investigating vulnerable dependency through a comparative study of existing security tools*. With that goal, we ask:

> RQ: **How do the analysis results of existing vulnerable dependency detection tools differ in comparison to each other?**

To answer, we choose OpenMRS, a web application for Electronic Medical Platform (EMR), as our evaluation subject. As a large software system with more than 2,500 third-party dependencies, OpenMRS comprises of 44 separate projects primarily written in Java and JavaScript and uses Maven and NPM as package managers for the respective languages. We scan OpenMRS by 10 industry leading VD detection tools covering both open source and commercial offerings. The tools vary in the techniques they

[1] http://www.modulecounts.com/

apply to scan the dependencies, e.g. analyzing build file and binaries, static and dynamic analysis of the code; in their vulnerability data source, and the additional information they provide in order to help developers prioritize fixes. We study the differences in these tools' analysis, categorize how and why their results differ, and discuss the strength and weaknesses of these tools. **[[revise after finishing findings]]**

We make following **contributions** in this paper:

1. **[[TODO]]**

[[Talk about paper structure:]] Section 2 discusses background knowledge and terminologies used in the paper while section 3 discusses related work. Section 3 explains the evaluation software, and Section 4 explains the VD detection tools we have, and how we perform the analysis for each of them. From Section 5, Findings start.

2 Background & Terminologies:

Below, we discuss some common terminologies as used in this paper:

Package Manager: A package manager or package-management ecosystem is a collection of software repositories that hosts open source packages along with all their historical versions with a unique tracking system. The open source projects, however, are maintained by respective project developers. The package manager also offers a collection of software tools that can automate the process of both uploading the packages for project maintainers; and download, installation, updates, and removal of the packages for the users in their host machine. Maven, NPM, RubyGems are the most popular package managers for handling for Java, Node.js (JavaScript), and Ruby open source packages respectively.

Dependency: When a software uses (depends on) on a third-party package (usually open source) for some functionality, the package is referred to as a *dependency* of the software. Typically, the software declares a specific version (or, a range of valid versions) of the open source package as its dependency. In the remainder of this paper, we refer to 'dependency' as a specific version of a package in general. For e.g. version 1.0.0 and version 2.0.0 of same package **A** will be considered as distinct dependency when we discuss our findings in this paper. However, they will be considered as the same package.

A software can use and distribute it's dependency in different ways. The end-user of the software may receive a dependency compiled with the software or, may be individually responsible for providing the dependencies to run the software. Moreover, the software may use a dependency only during development or testing or also in its production environment. When and how a dependency is used by a software is determined by its *scope* by the respective package manager.

Dependency File: The list of required dependencies of a software is usually declared in a separate file along with other metadata of the software. A package manager reads this file to identify the required dependencies (and the transitive dependencies required to run the dependencies), and can automate the installation process for the user. For Java and Maven, the file is names as *pom.xml*; while for node.js and NPM, the file is named as *package.json*. In this paper, we refer to these files that contain dependency information and are used by the respective package manager, as *dependency file*.

Dependency Tree: A software lists its dependencies functionality of which it intends to use directly from its own code. Such dependencies are called *direct dependencies*. However, the direct dependencies may depend on other open source packages which is required by the host machine to run the software successfully. The dependencies of direct dependencies , therefore, also dependencies of the software, and called *transitive dependencies*. Therefore, for most package managers including Maven and NPM, the whole dependency structure is hierarchical, and can be viewed as a tree format. However, dependency conflicts may arise in different scenarios such as when different packages depend on different versions of the same package. Package managers resolve dependency conflicts differently and structure the final dependency tree according to their own policies. *Depth* of a dependency refers to their level in the dependency tree with direct dependencies having depth of one.

Vulnerability & CVEs: A software vulnerability is a flaw in the software that may be exploitable in order to breach the confidentiality, integrity, and availability of the software. Security researchers and developers constantly discover new vulnerabilities in already released versions of software packages, and can report the vulnerability to the respective project maintainers or any central vulnerability tracking authority so that the maintainers can fix the vulnerability (i.e *patch*) and release a new version of the software; and the users of the software can be aware of the vulnerability and update to the fixed version or just apply the patch.

The most common vulnerability reference-tracking system is Common Vulnerabilities and Exposure (CVE) where each vulnerability is referenced by a unique CVE identifier. US National Vulnerability Database (NVD) is the most well-known database to maintain CVEs. However, there are other vulnerability databases that tracks vulnerabilities not necessarily having a CVE identifier, such as NPM Security Advisories, GitHub security advisories, and other security tools with own proprietary vulnerability database. Vulnerabilities are typically assigned with a severity rating. While the Common Vulnerability Scoring System (CVSS) is the industry standard for assessing severity, databases like GitHub and NPM has their own severity ratings. In this paper, the vulnerability that does not have an associated CVEs are referred to as *Non-CVEs*.

Our evaluation subject, OpenMRS, contains projects that uses Maven package manager for Java projects and NPM for JavaScirpt projects. Below, we briefly describe the dependency scopes and dependency mediation as maintained by these two package managers:

2.0.1 Maven Dependency system:

Below are the different dependency scopes under Maven [18]:

1. Compile: Compile dependencies are available in all classpaths of a Java project and propagates to dependent packages as transitive dependencies. This is the default scope in maven.
2. Provided: A dependency that is required during compilation and testing but is not added to the Java project's runtime classpaths. Therefore, the user has to provide this dependency in runtime. Provided dependencies do not propagate as transitive dependencies.
3. Runtime: This scope indicates that the dependency is not required for compilation, but is required for execution. Maven includes dependency with this scope in the runtime and test classpaths, but not compile classpaths. Runtime dependencies propagate as transitive dependencies.
4. Test: The dependency with this scope is not required for normal use of the software, and is only required in the testing phases. The scope is not transitive.
5. System: This scope is similar to provided except that user has to explicitly provide the dependency jar file (not automatically managed by the package manager).

Dependency Mediation: When there are multiple versions of a package in the dependency tree, Maven picks the nearest definition – that is – the version on the smallest depth and the one that is first declared among equal depths. Therefore, usually a single project has a single version of a package as dependency. Maven stores the dependencies in a local cache on the host machine from where they are read. Therefore, when multiple projects use the same dependency, usually the dependency is read from the same source.

In the Maven dependency file (*pom.xml*), developers generally specify a single version for its dependencies, or can refer to the latest release through several keywords.

2.0.2 Node Package Manager

Below are the different dependency scopes under NPM:

1. Production (Prod): Packages required by the software application in production.
2. Development (Dev): Packages that are only required for local development and testing.
3. Peer: Peer dependencies are not automatically installed by the package manager as transitive dependencies, but may require to be installed by the dependant software as a *peer* for some functionality. Typically, when the dependant application and a dependency both uses another package and requires a some for of communication among them, the user needs to install the package explicitly as a direct dependency to ensure same versioning and dependency path.
4. Optional: The dependency is not required to run its core functionality and not automatically installed by the package manager.

Dependency Mediation: NPM copies all the dependencies within the project directory in a sub-directory called 'node_modules' where the dependencies are copied in the similar structure of the dependency tree. Therefore, if package A and B both depends on version 1.0.0 and version 2.0.0 of the same package C, two different copies of package C of respective versions will be copied inside the directories of package A and B. While the process may create duplicate copies of packages, it reduces chances of any version conflict within dependencies. However, due to this process, same dependency of a project (package and version) can have multiple ***dependency path*** (the path through they are introduced to the root application). Therefore, when scanning vulnerable dependencies, *dependency path* is an important concept for NPM dependencies.

In the NPM dependency file, developers can list a range of versions that is valid as a dependency for a specific package. NPM also has the concept of *lock* files – a snapshot of the entire dependency tree and their resolved version at a given time; and can be used to instruct NPM to install the specified versions in the lock file.

3 Related Work

- Vulnerable Dependency: Talk about dependency graph in general. Decan et al. [9] analyzes library.io dataset and study's package ecosystems. Then, go onto talk about vulnerability in dependency. What is the impact? Decan et al. [8], and work from sabetta group. Mention equifax here.

  Kuala et al. [15] talks about how developers update dependencies and if security advisory plays any role. Talk about our FSE Paper.

  What tools are there? What is recent research? mainly SourceClear and Vulas? Mention there paper.

- Evaluation of security tools. How these are done and what are the generic findings? **[[Prior example of tool evaluation or comparison work?]]** SATE paper. Recent paper on comparison of technical debt measuring tools.

- **[[Security research on measuring exploitability and/or risk of a vulnerability. Please refer me good papers]]**

4 Evaluation subject software: OpenMRS

OpenMRS is an Electronic Medical Record (EMR) platform, established in 2004 [2]. The platform meets diverse use cases from medical appointment, research database, to tracking patients at sporting events. The platform is structured as a modular, multi-layered system and can be used in many different configurations. A particular configuration of OpenMRS that can be installed and upgraded as a unit is referred to as a "distribution". The general purpose distribution of OpenMRS is the "Reference Application Distribution" [2]. Implementers and developers can build new modules and add functionalities to this distribution in order to customize the EMR facility specific to their needs. In this paper, we choose version 2.10.0 of "OpenMrs Reference Application Distribution" released on April 6, 2020 (latest release at the time of this study) as our case study to evaluate the existing VD detection tools. In the remainder of the paper, we refer to this distribution simply as "OpenMRS".

OpenMRS distribution consists of 44 projects that are hosted in their own separate repositories on GitHub. While these projects are called as *modules* that makes up the whole distribution, these modules themselves can have further modular design (sub-projects). In this study, we consider each project hosted on its own repository as an individual entity regardless of their structure.

39 out of the 44 projects are Maven packages and 1 project is an NPM project. Rest of the 4 projects contain both Maven and NPM package, one each. Given the complicated structure of these projects, we further analyzed them with 'github-linguist' [3] tool to determine all the programming languages these projects contain code in. Besides Java and JavaScript, the other languages are: Ruby, HiveQL, CSS, Groovy, HTML, XSLT, AngelScript, Shell, SQLPL. Only one project contains some Ruby code (0.8% of the repository) with two Gemfiles. We ignore Ruby in this study due to its minuscule amount. The other languages do not have an explicit package dependency ecosystem. Therefore, for the evaluation of VD detection tools in this paper, we focus on analyzing vulnerable dependencies for Java and JavaScript code from OpenMRs.

4.1 Why OpenMRS?

Choosing test cases to evaluate software security tools is not a straightforward task. The challenges of test case selection are discussed in [10]. The paper discusses three characteristics for an ideal test case candidate: 1) representative of real, existing software; 2) sufficient and diverse number of security weaknesses that are being analyzed; and 3) availability of ground truth. Since, ours is the first formal study evaluating VD detection tools, we have no existing benchmark to follow. There is one evaluation framework available prepared by a commercial VD detection tool, named 'Evaluation Framework for Dependency Analysis' [1]; however, the framework consists of artificially synthesized projects and being non peer-reviewed, might be biased to the representative tool.

Delaitre et al. in their paper [10] recognizes that no candidate test case exhibits all three characteristics of an ideal; and opines that any software system meeting two of three characteristics can be a valid candidate. OpenMRS, a large software system, that is actively used and maintained is a *real, existing* software. Moreover, the software system consists of 44 GitHub projects and contain code maintained in Maven and Node.js ecosystem, two of the most popular and large package ecosystems. Consequently, the project depends on a large number of third-party library and packages, as seen in Section 4.2, which increases the probability of having *sufficient, diverse number* of vulnerable dependencies as security weaknesses. While there is no ground truth available and we will not be comparing the VD detection tools in terms of precision or recall metrics, large software system having many third-party dependencies makes OpenMRS a suitable candidate for this study.

Another approach of evaluating VD detection tools can be running the tools on a group of selected projects from diverse background rather than a single case study. However, some of the selected VD detection tools in this study are resource and time-consuming to run while some tools involve sophisticated analysis which have certain requirements (e.g. integration/acceptance testing for interactive testing, unit testing for executablity tracing) and complicated set-ups that creates a barrier to scale up our study to projects of diverse structure and configuration. Moreover, focusing on a single case study enables us to manual investigation for qualitative analysis of the tools' analysis results. Finally, OpenMRS consists of 44 projects. Although these projects belong to the same organization, they may have different maintenance and development team and practices which gives some variation on how dependencies are maintained within the studied projects.

OpenMRS, as an elaborate open source software system, has also been used in security and privacy research in the past [4, 6, 7, 16, 20, 22], as well as other similar medical platforms [5, 11]. Lamp et al. [16] evaluated OpenMRS for several medical system security requirements; Rizvi et al. [20] evaluted OpenMRS for access control checking; while Amir-Mohammadian et al. [4] studied OpenMRS for correct audit logging in the past.

[2] http://guide.openmrs.org/en/
[3] https://github.com/github/linguist

	Java	JavaScript
No. of projects	43	5
Total unique dependencies (package and version)	547	2,213
Total unique packages	311	1,498
Median dependency per project	127.0	840.5
Median dependency path per project	NA	1,675.0
Median depth of dependencies	2	4
Max depth of dependencies	7	12
Median Provided dependencies	99.0	NA
Median Compile dependencies	3.0	NA
Median Runtime dependencies	5.0	NA
Median Test dependencies	24.5	NA
Median Production dependencies	NA	202.5
Median Production dependency path	NA	366.0
Median Developer dependencies	NA	807.5
Median Developer dependency path	NA	1,613.5

Table 1: OpenMRS Dependency Overview

4.2 OpenMRS: Dependency Overview

In this section, we provide an overview of dependencies for Maven (Java) and npm (JavaScript) packages for OpenMRS. We parse dependency tree of each of these packages by native 'mvn dependency:tree' and 'npm list' command. For each project, A dependency is defined as a unique version of a package. We also parse each dependencies scope and depth in the dependency tree.

In Table 1, we show total count of unique dependencies for OpenMRS for both Java and JavaScript. Then, we show a breakdown per project: median dependency count, median dependency path in case of NPM, median and maximum depth of a dependency. We also provide a dependency count breakdown per scope for both Maven and NPM. Note that, for Maven projects, there can be first-party dependencies for a project – that is – other projects from OpenMRS organization and the reference application distribution can be listed as a dependency. We did count the first-party a dependency as a dependency as they are part of the full software system. Additionally, for NPM projects, we discarded *peer* and *optional* dependencies as they are not automatically resolved and installed by the host package manager.

Note that, one NPM project has a *npm-shrinkwrap.json* file that provides specific version for each package in the dependency tree required by the software; and another NPM project has a *bower.json* file that is installed via NPM and can act as an individual package manager for front-end JavaScript packages. While we do not consider these files to generate dependency information in Table 1, a VD detection tool may consider them in analysis.

5 Vulnerable Dependency Detection Tools

To identify the existing VD detection tools from both industrial offering and the latest research, we performed an academic literature search and a web search with the following query: ("vulnerable" OR "open source" OR "software") AND ("dependency" OR "package" OR "library" OR "component" OR "composition") AND ("detection" OR "scan" OR "tool" OR "analysis"). From the relevant search results, we filtered the tools with following *exclusion criteria*: a) does not scan neither Maven nor NPM packages; b) offers an executable tool and we have access to the tool; c) does not offer any unique feature in vulnerability data source or scanning techniques than already selected tools and had relatively lesser web presence - we added this exclusion criteria in order to not overcrowd our tool selection and complicate the analysis process.

With our selection process, we selected ten VD detection tools. We selected Steady and Commercial A based on relevant academic literature on the tool, and the rest from web search. For three commercial tools that were not freely available or only available with a free trial, we used this tool with permission from the respective tool authority with the condition to not express the tools' name.

Table 2 presents an overview of the selected tools – their vulnerability data source and scanning techniques. Below we describe how we performed the scans for each of these tools.

1. **OWASP Dependency-Check** OWASP dependency-check can run for all languages.(but javascript is still run under experimentalAnalyzer setup). While we can enable additional analyzer and data scource for e.g. retirejs for javascript, we run the default scanning with experimental analyzer enabled.

2. **Snyk** Snyk is a VD detection tool [4] that finds vulnerable open source dependencies for all major programming languages including Java and JavaScript. The tool maintains their own vulnerability database. We use the command line tool of Snyk v1.382.0. We use the command 'snyk test –all-projects –dev –json' to get the report. It shows CVSS scores both for cves and non-cves and gives a severity level based on that. https://support.snyk.io/hc/en-us/articles/360001040078-How-is-a-vulnerability-s-severity-determi Although critical is still counted as high.

3. **Maven Security Versions** Maven Security Versions [5] is a VD detection tool solely for Maven package ecosystem maintained by the Victims project. The tool uses a vulnerability database also maintained by the Victims project. For the 43 Maven projects, we ran this tool by the maven command 'mvn

[4] https://snyk.io/product/open-source-security-management/
[5] https://github.com/victims/maven-security-versions

Tool	Java	JavaScript	Techniques
OWASP Dependency-Check	✓	✓	Third-part vulnerability database; analyzes dependency files either as maven plugin or scanning dependency file
Snyk	✓	✓	vulnerability database; analyzes dependency files from project root directory
GitHub Dependabot	✓	✓	Own vulnerability database; analyzes dependency files native GitHub service for hosted repositories
Maven Security Versions	✓		Own vulnerability database; analyzes dependency files through maven plugin
NPM Audit		✓	Own vulnerability database; analysis by scanning dependency files; provides list of fix actions
Steady	✓		Research tool; Fixed vulnerability data set with additional information on patch and vulnerable ; performs static call graph construction, executability tracing from unit testing, and bytecode instrumentation from integration testing in order to provide vulnerability reachability information from the software application
WhiteSource	✓	✓	Own vulnerability database; GitHub integration app
Commercial A (SourceClear)	✓	✓	Own vulnerability database; performs static code analysis to provide vulnerability reachability information
Commercial B (Seeker)	✓	✓	Binary analysis
Commercial C (Contrast)	✓		Identifies dependency in use through binary instrumentation during interaction or integration testing

Table 2: Vulnerable Dependency Detection Tools

com.redhat.victims.maven:security-versions:check‘ and parsed the report generated in ‘html’ format. For the multi-module projects, it may show the same alert multiple times for different modules, we deduped them. It does not give any severity levels.

4. **NPM Audit** NPM's own. Has automatic fix option. Which can work Run audit fix without modifying node-modules, but still updating the pkglock:
5. **GitHub Dependabot**
6. **Steady**

6 Findings: RQ1

6.1 Strengths & Weakness

Here, we will focus on mainly the alerts and the information generated with it but not about the user interface or presentation formats.

- Duplication of alerts for multi-module maven projects.
- We can enable and disable certain alerts
- Snyk not only provides which versions a vulnerability is fixed in, but also the functions with its filepath that was involved in the vulnerability and the new functions. Also reference links.
- Github dependabot exactly which version being used is vulnerable. That can be problematic in dep tree like npm that contains multiple versions of same package.
- For complex project structures, for e.g. multi-module maven with another javascript project in a sub-directory, most tools like OWASP fails to detect all of them. Snyk was perfect in this scenario. We had to manually manouver owasp to get them.
- OWASP has confidence and evidence count. It also detects javascripts components even in pure maven projects. OWASP also detects flaws in build file like pom.xml itself.(uicommons-scss@2.12.0)
- NPM Audit can have same CVE listed under more than one vulnerabilities in it's own database based on who reported and how it is described. For e.g., "CVE-2019-10744" is listed under both id 1523 and 1065. NPM AUdit also has their own severity rating with an exploitability score and a severity ratins. TODO: present a comparison between its severity rating and CVSS scores. Also, GitHub's ?
- NPM audit only tells if major change or not by looking at semver format. no distinction between minor change and patch. Snyk also tells if patchable or not. For the ones with no patch/fix available, npm audit gives information to review for project developers. Unable to find out multi-project structures.
- How github parses transitive deps. For npm, only when lock file available? check! Github found dependencies from old package-lock.json file that npm no longer lets us to worki with ‘Eintegrity’ error. So when we remove lock file and do npm install again we don't find this dependency.

6.2 For each tool, why the count of alert and distinct vulnerable dependency/ dependency path differ?

6.3 How many vulnerability since release?

7 Future Thoughts

- It is important to know if my application uses the vulnerable code both in order to understand if it is exploitable and if the patch has any chance in breaking code. Also, if patch touches more than the vulnerable part of the code then we need to check if all those code is reachable from my application or not on order to understand if it will break my changes.

References

[1] Evaluation framework for dependency analysis.

[2] Openmrs reference application distribution.

[3] 0patch.com. Security patching is hard.

[4] Sepehr Amir-Mohammadian, Stephen Chong, and Christian Skalka. Correct audit logging: Theory and practice. In *International Conference on Principles of Security and Trust*, pages 139–162. Springer, 2016.

[5] Andrew Austin and Laurie Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *2011 International Symposium on Empirical Software Engineering and Measurement*, pages 97–106. IEEE, 2011.

[6] Steven P Crain. Open source security assessment as a class project. *Journal of Computing Sciences in Colleges*, 32(6):41–53, 2017.

[7] Beatriz Sainz de Abajo and Agustín Llamas Ballestero. Overview of the most important open source software: analysis of the benefits of openmrs, openemr, and vista. In *Telemedicine and e-health services, policies, and applications: Advancements and developments*, pages 315–346. IGI Global, 2012.

Tool	Alert	Unique Dependency	Unique Package	Unique Vulnerability	CVE	Non-CVE	Scan Time (Minutes)
	Total (Median per project)						
Maven Security Versions	3,197 (58.0)	36 (12.0)	14 (12.0)	36 (22.0)	36	0	2.5
Snyk	4,968 (62)	96 (6.0)	46 (6)	186 (22.0)	175	11	16.3
GitHub Dependabot	136 (0.0)	20 (0.0)	11 (0.0)	61 (0.0)	61	0	NA
OWASP Dependency-Check	12,833 (253.0)	354 (38.0)	181 (37)	510* (119)	286	224*	10.7
Contrast	57	17	17	57	57	0	NA

Table 3: Vulnerable Dependencies for Java project

Tool	Total	Provided	Compile	Runtime	Test
	Unique Dependency (Unique Vulnerability)				
Maven Security Versions	36 (36)	35 (35)	6 (9)	0 (0)	5 (9)
Snyk	96 (186)	75 (181)	40 (97)	1 (2)	21 (22)
GitHub Dependabot	20 (61)	8 (43)	11 (60)	1 (1)	2 (2)
OWASP Dependency-Check	354 (510)	80 (265)	31 (131)	3 (13)	2 (60)

Table 4: Scope Breakdown

Tool	Alert	Unique Dependency Path	Unique Dependency	Unique Package	Unique Vulnerability	CVE	Non-CVE	Scan Time (Minutes)
	Total (Median per project)							
Snyk	2100 (76)	954 (40)	89 (19)	53 (16)	119 (26)	77	42	1.0
GitHub Dependabot	48 (4)	NA	31 (1)	29 (1)	44 (4)	28	16	NA
OWASP Dependency-Check	677 (60)	236 (10)	190 (10)	134 (7)	219* (37)	51	168*	4.43
NPM Audit	1229 (37)	815 (28)	55 (11)	43 (11)	60 (15)	30	30	?

Table 5: Vulnerable Dependencies for JavaScript project

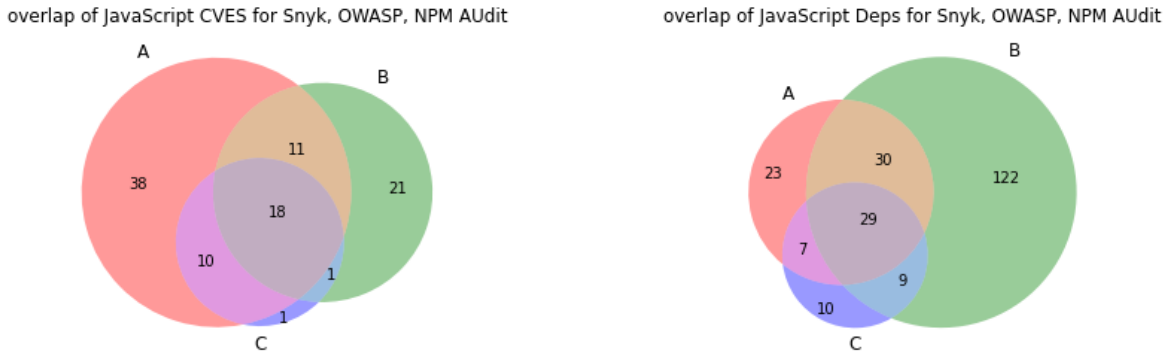

Figure 1: Overlap of Dependencies and CVEs for JavaScript tools

Figure 2: Overlap of Dependencies and CVEs for Java tools

Tool	Total	Prod	Dev
Snyk	89 (119)	12 (24)	83 (105)
GitHub Dependabot	31 (44)	5 (9)	8 (9)
OWASP Dependency-Check	190 (219)	57 (52)	167 (168)
NPM Audit	55 (60)	13 (10)	52 (57)

Table 6: Scope Breakdown

[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 181–191, 2018.

[9] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[10] Aurelien M. Delaitre, Bertrand C. Stivalet, Paul E. Black, Vadim Okun, Terry S. Cohen, and Athos Ribeiro. SATE V Report: Ten years of static analysis tool expositions. Technical report, National Institute of Standards and Technology, 2018.

[11] Maryam Farhadi, Hisham Haddad, and Hossain Shahriar. Static analysis of hippa security requirements in electronic health record applications. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 474–479. IEEE, 2018.

[12] Josh Fruhlinger. Equifax data breach faq: What happened, who was affected, what was the impact?, 2020.

[13] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 323–333. IEEE, 2019.

Language	Total Vuln.	Fix available	Patch avaialble	Exploit avialable
Java	192	139	0	TODO
JavaScript	120	67	16	TODO

Table 7: Vulnerability Infos from Snyk

[14] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249. IEEE, 2019.

[15] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[16] Josephine Lamp, Carlos E Rubio-Medrano, Ziming Zhao, and Gail-Joon Ahn. The danger of missing instructions: a systematic analysis of security requirements for mcps. In *2018 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*, pages 94–99. IEEE, 2018.

[17] Top OWASP. Top 10-2017 the ten most critical web application security risks. *OWASP_Top_10-2017_% 28en*, 29, 2020.

[18] Apache Maven Project. "introduction to the dependency mechanism".

[19] Teri Radichel. Why patching software is hard: Technical challenges.

[20] Syed Zain Rizvi, Philip WL Fong, Jason Crampton, and James Sellwood. Relationship-based access control for openmrs. *arXiv preprint arXiv:1503.06154*, 2015.

[21] Help Net Security. Surge in cyber attacks targeting open source software projects.

[22] Inger Anne Tøndel, Martin Gilje Jaatun, Daniela Soares Cruzes, and Laurie Williams. Collaborative security risk estimation in agile software development. *Information & Computer Security*, 2019.

[23] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, pages 1–26, 2012.

[24] Jim Witschey, Shundan Xiao, and Emerson Murphy-Hill. Technical and personal factors influencing developers' adoption of security tools. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, pages 23–26, 2014.

[25] Zeljka Zorz. The percentage of open source code in proprietary apps is rising.