More May Not Necessarily Be Better: A Comparison of Vulnerable Dependency Detection Tools

Abstract

Modern software uses many third-party libraries and frameworks as *dependencies*. Known vulnerabilities in dependencies is a major security risk. While tools exist to detect vulnerable dependencies, a formal study is yet to be performed to evaluate the existing tools. The goal of this study is *to aid security practitioners and researchers in understanding the current state of vulnerable dependency detection through a comparative study of existing tools.* We run 10 industry-leading dependency detection tools on a large web application composed of Maven (Java) and npm (JavaScript) projects. We find that the tools' results vary for both vulnerable dependencies and the unique vulnerabilities. The count of identified dependencies with known vulnerabilities ranges from 17 to 332 for Maven and from 32 to 239 for npm projects across different tools. Similarly, the count of unique vulnerabilities ranges from 36 to 313 for Maven and from 45 to 234 for npm projects. However, none of the tools include all the findings from the other tools. We find that inconsistency in vulnerability to dependency mapping is a primary reason behind differences in tools' results. Further, tools can provide additional metrics to assess the risk of a vulnerability from the context of the dependant application. We provide a characterization of such metrics as provided by different tools.

1 Introduction

Modern software typically uses 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 dependency packages [52]. A 2017 study found that 96% of the commercial applications contain open source packages with an average of 257 packages per application [55]. The package manager for the JavaScript programming language, npm, hosts more than 1.3 million open source packages at the time of this writing [9].

However, known vulnerabilities in dependencies is one of the top ten security risks [39]. 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 until July as Equifax kept using the vulnerable version of the framework [30]. Since this incident, a recent security report has found a 430% surge in cyber attacks aimed at upstream open source packages [48].

Several security tools, both open source and commercial, exist that detect the dependencies of a software with known vulnerabilities, i.e. *vulnerable dependencies* (VD). These tools differ in how they detect the used dependencies and how rich and accurate is their vulnerability database. Therefore, the results from different tools can vary from each other. A formal study is yet to be performed to compare the techniques and the scan results of the existing tools.

Further, not all alerts generated by security tools are relevant or high-priority to the developers [32,41]. False positives are a common problem for security tools [33]. The tools can report vulnerabilities on dependencies that are never actually used in production or the vulnerability can be in parts of dependency not used by the product. Also, vulnerability data may spread across different databases, bug repositories and tools may lack in completeness of their reports.

Assessing the risk of vulnerabilities in dependency is also necessary from a prioritization point. The fix of the VDs can have high cost due to many factors, including but not limited to, regression testing; breaking changes; and application legacy [22,46]. Severity rating of the vulnerability may not be adequate and developers would need to assess risk from the context of the dependant application. What additional information is presented by existing tools to aid in such risk assessment has not been formally characterized yet.

The goal of this study is *to aid security practitioners and researchers in understanding the current state of vulnerable dependency detection through a comparative study of existing tools.* We answer the following research questions:

RQ1: **How do the analysis results of existing vulnerable dependency detection tools differ in**

comparison to each other?

RQ2: **What additional information is presented by the existing tools to aid in assessing the risk of vulnerability in dependencies?**

To answer, we evaluate 10 vulnerable dependency detection tools on a large web application, OpenMRS, composed of Maven (Java) and npm (JavaScript) projects. The selected tools vary in their scanning technique and vulnerability data source and represent the state-of-the-art while the evaluation subject covers two of the most popular package ecosystems. The contributions of this paper are:

1. A comparative analysis of ten VD detection tools over Maven and npm dependencies;
2. A manual analysis of differences in tools' results;
3. A characterization of additional risk assessment metrics provided by the tools.

The remainder of the paper is structured as follows: Section 2 introduces the key concepts and terminologies; Section 3 and 4 explains the evaluation subject software and the studied tools. Section 5 and 6 discusses the findings of this paper followed by discussion and limitation of the findings. Section 9 discusses related work followed by conclusion.

2 Key Concepts & Terminologies:

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

Package Manager: A package manager is a collection of software tools that hosts open source packages and automates the process of their configurations. Maven, npm, RubyGems are the most popular package managers for handling Java, Node.js (JavaScript), and Ruby packages respectively.

Dependency: When a software uses 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 third-party package as its dependency. In the remainder of this paper, we refer to 'dependency' as a specific version of a package. For e.g. version 1.0.0 and version 2.0.0 of the same package *A* will be considered as distinct *dependency*. However, they will be considered as the same *package*.

Dependency File: The list of required dependencies of a software is usually declared in a manifest file along with other project metadata. A package manager reads these files to resolve the required dependencies. *pom.xml* and *package.json* are dependency file for Maven and npm respectively.

Dependency Tree: The dependencies that a software accesses directly from its own code 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. Such packages are called ***transitive*** dependencies. Therefore, for most package managers, including Maven and npm, the whole dependency structure is hierarchical and forms a tree format. ***Depth*** of a dependency refers to their level in the dependency tree with direct dependencies having depth of one.

Vulnerability: NIST [38] defines vulnerability as a "weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source." Security practitioners constantly discover new vulnerabilities in already released versions of software packages. If reported, respective project maintainers can fix the vulnerability in a new version.

Vulnerable dependency (VD): When a dependency of a software is subject to publicly known vulnerabilities, it is referred to as a VD. To mitigate the threat, the maintainers of the software can either remove the dependency or change the dependency to a safer version.

Common Vulnerabilities and Exposures (CVE): The primary reference-tracking system for known vulnerabilities is Common Vulnerabilities and Exposure (CVE) system where each vulnerability is referenced by a unique CVE identifier. There are publicly accessible databases for CVEs such as NVD [10], Mitre [8]. However, there are databases that track known vulnerabilities not necessarily having a CVE identifier, such as npm Security Advisories [11], Sonatype OSS Index [17], GitHub security advisories [6]. Similarly, security tools can also maintain their own vulnerability databases. In this paper, the vulnerability that does not have an associated CVE identifier is referred to as ***Non-CVE***.

2.1 Maven:

Maven is a package manager for Java projects.

Dependency Scopes: Maven dependencies can have six different scopes [45]: compile, provided, runtime, test, system, and import. The scopes determine which classpaths a dependency will be available in (compile, test, and/or runtime) and if the dependencies can propagate transitively..

Dependency Mediation: When there are multiple versions of a package in the dependency tree, Maven picks one with the nearest definition. Therefore, usually a single project has a single version of a package as a dependency. Maven stores the dependencies in a local cache on the host machine from where they are read. In the dependency file (*pom.xml*), developers generally specify a single version for its dependencies.

Version numbering: Version numbers can have up to five parts indicating major, minor, or incremental changes.

2.2 Node Package Manager (npm):

npm is a package manager for JavaScript projects.

Dependency Scopes: npm has two primary dependency scopes: *Prod* (production) and *dev* (development) to indicate the phase where a dependency is required. Two other dependency scopes, peer and optional, are not automatically resolved by npm and therefore, are ignored in this study.

Dependency Mediation: npm copies all the dependencies in a project sub-directory called 'node_modules', with the similar structure of the dependency tree. If two dependencies *A* and *B* both depends on the same package *C*, two different copies of package *C* will be copied inside package *A* and *B*. Therefore, the same dependency can have multiple paths to be introduced to the root application. Also, the same package can have multiple versions as dependencies. Therefore, npm has a concept called ***dependency path*** which is not present in Maven. Each unique path a dependency is introduced to the root application is referred to as *dependency path*.

In npm, developers can list a range of versions for a package that is valid as a dependency. 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.

Version numbering: npm packages follow the SemVer format [1] for version numbering which has three parts indicating a) major release (may break backward compatibility), b) minor release (backward compatible new features), c) patch release (backward compatible bug fixes).

3 Evaluation subject software: OpenMRS

OpenMRS is a web application for electronic medical record platform [12]. The application is structured as a modular, multi-layered system. 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" [13]. We choose Version 2.10.0 of this distribution released on April 6, 2020 (latest release at the time of this study) as our evaluation subject. In the remainder of the paper, we refer to the whole distribution simply as "OpenMRS".

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

Out of the 44 projects, 39 are Maven projects and 1 is an npm project. The other 4 projects are composed of a Maven and an npm project each. We identify Maven and npm project based on the presence of *pom.xml* and *package.json* file specifying metadata for corresponding projects. Therefore, we scope our study to Maven and npm dependencies..

OpenMRS provides a development toolkit, OpenMRS SDK [14], to automate the build, test, and run of the individual projects and assemble the full application. We use this SDK in this study to perform the required security testing.

3.1 Why OpenMRS?

Choosing test cases to evaluate software security tools can be a complex task. The challenges of test case selection discussed in [29] include three characteristics for an ideal test case candidate: 1) representative of real, existing software; 2) sufficient and diverse number of security weaknesses; 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 VD detection tool [5]. However, this non peer-reviewed framework consists of artificially-synthesized projects and might be biased to the representative tool.

Delaitre et al. [29] recognizes that no candidate test case exhibits all three characteristics of an ideal; and opines that any software system meeting two of the three characteristics can be a valid candidate. OpenMRS, being actively used and maintained, is a *real, existing* software. Moreover, the software contains Maven and npm projects – two of the most popular and large package ecosystems. Consequently, the software depends on a large number of third-party dependencies as seen in Section 3.2; and being a web application, is composed of several heterogeneous components which increases the probability of having a *sufficient, diverse number* of VDs.

Regarding the third characteristic, we have no ground truth available for OpenMRS. Determining *true positives* for VDs has no established guidelines and can be subjective. However, we will not be comparing the VD detection tools in terms of precision or recall metrics; our goal is to only measure the differences in the tools' results. Therefore, having met the two of the three characteristics mentioned in [29], OpenMRS is a suitable candidate for this study.

Another approach of evaluation instead of a single case study can be running the tools on a group of diverse projects. However, some of the selected tools in this study are resource and time-consuming to run while some tools involve sophisticated analysis with certain requirements (e.g. acceptance tests for interactive binary instrumentation, unit tests for executablity tracing) and set-up that creates a barrier to scale our study to projects of diverse structures and configurations. Moreover, focusing on a single case study enables us to manually investigate the differences in the tools' results.

OpenMRS has also been used in security and privacy research in the past [23, 25, 26, 36, 47, 51]. Lamp et al. [36] evaluated OpenMRS for several medical system security requirements; Rizvi et al. [47] evaluated OpenMRS for access control checking; while Amir-Mohammadian et al. [23] studied OpenMRS for correct audit logging.

3.2 OpenMRS: Dependency Overview

In this section, we provide an overview of Maven and npm dependencies of OpenMRS. We parse the dependency tree of each project through native `mvn dependency:tree` and

Table 1: OpenMRS dependency overview

	Maven	npm
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

`npm list` command. We also parse each dependency's scope and depth in the dependency tree.

Table 1 provides a dependency overview of OpenMRS. Note that, for Maven projects, there can be first-party dependencies – that is – a project within the OpenMRS distribution can listed as a dependency for another project. We do not count the first-party dependencies in Table 1. Also, npm projects can contain *lock* files such as *shrinkwrap.json*, *package-lock.json* which are not considered in Table 1.

4 Vulnerable Dependency Detection Tools

4.1 Tool Selection

To identify the existing VD detection tools from both industrial offerings and the latest research, we performed an academic literature search and a web search through following keywords: (*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 *inclusion criteria*: a) scans either Maven or npm projects; b) we have an access to an executable tool; c) offers unique features to already selected tools. From our selection process, we selected ten tools. Three of the tools are not freely available and license agreement prevents us from providing the names. We refer to them as Commercial A, B, and C.

We observed that tools primarily differ in three dimensions:

1. **Vulnerability data source:** To provide the list of known vulnerabilities, the tools need a data source. Tools can pull vulnerability data from *third-party* source(s) that includes public databases like NVD CVEs. Also, tools may have a ***self-curated*** database where they actively monitor and curate up-to-date vulnerability data. While the self-curated databases usually accommodate data from other public databases, tools may also have proprietary techniques to enrich and revise the vulnerability data with their own findings [24, 35, 54]

2. **Dependency scanning source:** The dependency files are the primary source to resolve dependencies of a project as is done by the package managers. However, tools can deduce dependencies from the source code and application binaries as well.

3. **Additional analysis to infer dependency use:** Tools can offer additional static and/or dynamic analysis to infer how the dependencies are being used.

Table 2 presents an overview of the ten selected tools based on the their documentation and our experience of using them. "No Information" is listed in the absence of accurate information. Tools can also suggest fix actions for the detected VDs. While we do not consider fix suggestions during tool selection, we discuss this feature in the findings.

4.2 Running the Tools

Below we describe how we performed the scans:

OWASP Dependency-Check (DC): This tool works by scanning the dependency files, JARs, and JavaScript files [7] and pulls vulnerability data from multiple third-party sources. We used the Maven plugin to scan Maven projects and the command line tool (Version 5.3.2) to scan npm projects. We had the *experimental analyzer* option enabled while running the tool as JavaScript is only available through this option.

Snyk: We ran the command line tool (Version 1.382.0) that is freely available. [15] through the command `snyk test -all-projects -dev -json`. The `-dev` flag ensures that *dev* dependencies will also get scanned in case of npm projects which is disabled by default.

GitHub Dependabot: We hosted the 44 studied projects on the first author's GitHub account and retrieved the Dependabot alerts through GitHub API. The projects were hosted at the release branch specified in the 2.10.0 version of OpenMRS distribution which we evaluate in this study.

Maven Security Versions (MSV): This tool is part of the Victims project [19] and is available for Java projects. We ran this tool through its Maven plugin.

npm audit: This is a native tool of npm package manager. We used the `npm audit -json` and the `npm audit -fix -dry-run -json` commands.

Eclipse Steady: This tool performs additional analysis to assess the execution of vulnerable code in the dependencies in a given application context [4]. The approach implemented is described in [43] and [42]. The tool requires a manual set

Table 2: Studied vulnerable dependency (VD) detection tools

Tool	**Java**	**JavaScript**	**Vulnerability Data Source**	**Dependency Scan Source**	**Additional Analysis**
OWASP Dependency-Check (DC)	✓	✓	Third-party: NVD, OSS Index, npm	Dependency file, source code, binary	
Snyk	✓	✓	Self-curated	Dependency file	
GitHub Dependabot	✓	✓	Self-curated	No Information	
Maven Security Versions (MSV)	✓		Self-curated	Dependency file	
npm audit		✓	Self-curated	Dependency file	
Eclipse Steady	✓		Self-curated	Dependency file, source code, binary	Static and dynamic analysis to determine reachability of vulnerable code
WhiteSource	✓	✓	Self-curated	No Information	
Commercial A	✓		Self-curated	No Information	Static and dynamic analysis to identify vulnerable call chains
Commercial B	✓		No Information	Binary	Interaction testing to identify dependency under use
Commercial C	✓		No Information	Binary	

up along with the vulnerability database. We used Version 3.1.10 of this tool. We set up Steady in a virtual machine allocating 16GB RAM and 4 processor cores. Steady hosts their vulnerability data set on GitHub [18]. The manually curated data set contains patch commit information for each vulnerability. We imported the data source updated on Jan 24,2020 in Steady. We then performed the patch analysis feature provided by the tool to identify the involved code constructs for each vulnerability.

For risk assessments of the identified VDs, Steady can perform three additional analysis: 1) static call graph construction; 2) executing JUnit tests for analyzing executability traces; and 3) JVM instrumentation through integration testing. We were unable to complete the third analysis as the tool presumably ran out of memory after running for ten days.

WhiteSource: WhiteSource has a GitHub app named "WhiteSource Bolt" [20]. We connected this app with our hosted repositories on GitHub and retrieved the issues created by WhiteSource through GitHub API.

Commercial A: This tool has scientific papers discussing their approach (not citing to maintain blindness). We contacted their research team and provided them with the repository links for the studied projects. They returned us with scan reports only for Maven dependencies for 37 projects and reported that they failed to complete the automated scans for the rest of the projects which may have required a manual intervention. This tool offers static analysis by default, and dynamic analysis as an option to identify vulnerable call chains. We received scan report only with static analysis performed on the code.

Commercial B: We used the free cloud edition of the tool. The tool scans dependency through interactive application security testing – that is – monitoring dependencies in use when an application is run and interacted with either through automated testing or human tester. OpenMRS provides 123 test cases for integration testing that interact with the application through a Selenium web-driver. We connected OpenMRS to this tool and used the integration test suite to interact with the application.

Commercial C: This tool connects with an application during runtime and scans the dependencies through the associated binaries. We set up a local server for this tool and connected to OpenMRS.

For eight of the ten tools, we collected the analysis report separately for 44 projects. For Commercial B and C, we get a single report for the whole OpenMRS distribution. As vulnerability data gets updated over time, we ran all the tools during the month of September 2020 to ensure a fair comparison with the exception of Steady which requires a manual setup. The vulnerability data in Steady is from January 2020.

4.3 Analyzing Tool Results

Below we discuss metrics and information that we processed from the tool reports to answer our research questions.

Quantity of Alerts: When a project is scanned by a tool, the tool reports a raw count of alerts identified on the project. However, the alerts do not represent either unique dependencies or unique vulnerabilities. The same alerts can be repeated due to OpenMRS's multi-module project structure. The alert count, however, may indicate the amount of effort required from the developers to audit the scan reports.

Table 3: Vulnerable Dependencies for Maven (Java) projects

Tool	Alert	Unique Dependency	Unique Package	Unique Vulnerability	CVE	Non-CVE	Scan Time (Minutes)
	Total (Median per project)						
OWASP DC	12,466 (254.0)	332 (38.0)	149 (36.0)	313 (117.0)	289	24	14.4
Snyk	4,902 (66.0)	96 (6.0)	46 (6.0)	189 (23.0)	178	11	15.1
Dependabot	136 (0.0)	20 (0.0)	11 (0.0)	61 (0.0)	61	0	NA
MSV	3,197 (58.0)	36 (12.0)	14 (12.0)	36 (22.0)	36	0	3.4
Steady	2,489 (51.0)	91 (20.0)	39 (19.0)	97 (41.0)	89	8	385.0
WhiteSource	434 (0.0)	76 (0.0)	44 (0.0)	146 (0.0)	127	19	NA
Commercial A	2,998 (70.0)	107 (24.0)	53 (24.0)	208 (70.0)	187	21	NA
Commercial B	205	35	35	127	127	0	NA
Commercial C	57	17	17	57	57	0	NA

Table 4: Vulnerable Dependencies for npm (JavaScript) projects

Tool	Alert	Unique Dependency Path	Unique Dependency	Unique Package	Unique Vulnerability	CVE, Non-CVE	Scan Time (Minutes)
	Total (Median per project)						
OWASP DC	1,379 (208.0)	498 (72.0)	239 (71.0)	160 (57.0)	234 (71.0)	78, 156	4.4
Snyk	2,210 (135.0)	1,004 (44.0)	90 (20.0)	54 (17.0)	121 (26.0)	79, 42	1.0
Dependabot	97 (8.0)	NA	32 (1.0)	30 (1.0)	45 (4.0)	29, 16	NA
npm audit	1,266 (37.0)	852 (28.0)	58 (12.0)	45 (12.0)	62 (16.0)	31, 31	0.1
WhiteSource	205 (32.0)	205 (32.0)	89 (14.0)	55 (9.0)	96 (18.0)	58, 38	NA

Tracking unique dependency, dependency path, package, and vulnerability: The definitions of these four metrics, as used in this study, are provided in Section 2. When processing the analysis reports from all the tools, we store the data in a relational database schema. In the schema, we keep an identifier for each unique package, dependency (package:version), dependency path, and CVE identifier. For the non-CVEs, all tools except OWASP DC and Commercial A provide a tool-specific identifier. While OWASP DC and Commercial A provide no reliable identifier to track unique non-CVEs, upon manual inspection, we noticed that vulnerability description along with the affected package(s) is a reliable way to track non-CVEs. However, we have no reliable way to map non-CVEs across different tool reports.

Scan time indicates the total number in minutes a tool took to scan all the projects. We have no scan time for GitHub and WhiteSource as they are GitHub cloud services. We collected the issues and alerts from GitHub at the end of September, at least two weeks after hosting the repositories. Both Commercial B and C detect dependencies during runtime, and therefore, have no definite scan time.

Other information: Tools had additional information in their report generally to aid developers in assessing the risk of the alerts and to help in fixing them. We also collected these additional data which will be explained in Section 5.3 and 6 when discussing the findings.

5 RQ1: How do the analysis results of existing vulnerable dependency detection tools differ in comparison to each other?

Table 3 and 4 shows the tools' result summary for Maven and npm dependencies, respectively. Note that, we only have partial scan reports (37 projects) for Commercial A. For eight tools that scanned projects individually, the tables show the median count per project for alerts and unique dependency, dependency path, package, and vulnerability besides the total count for the full application. The tables also report the total count of CVEs, non-CVEs, and scan time for each tool.

The alert counts are typically higher than the count of unique vulnerabilities or the count of vulnerable dependency paths. While the total alert count repeats the same vulnerabilities found across projects, many tools repeated the same alert for each sub-modules within a project as well. We also see the unique dependency count is higher than the unique package count. Different versions of the same package may be declared as a dependency in different projects for Maven while npm can have multiple versions of the same package even within a single project. Below we discuss our findings:

The tools vary in their results: We find a wide range

Table 5: Scope breakdown and Max. depth of detected vulnerable dependencies (VDs)

Tool	**Scope breakdown for Maven VDs**				**Scope breakdown for npm VDs**		**Max. Depth of VDs**	
	Compile	**Provided**	**Runtime**	**Test**	**Prod**	**Dev**	**Maven**	**npm**
OWASP DC	58	66	4	54	65	207	6	10
Snyk	56	62	2	25	13	83	7	10
Dependabot	15	5	1	2	6	8	2	6
MSV	19	30	1	3	NA	NA	5	NA
npm audit	NA	NA	NA	NA	15	51	NA	10
Steady	60	60	4	11	NA	NA	5	NA
WhiteSource	54	0	2	0	12	76	5	10
Commercial A	72	79	1	0	NA	NA	5	NA

across tools for both unique dependency and the unique vulnerabilities they found. OWASP DC detects the highest number of unique dependency and vulnerabilities. Conversely, Commercial B and C, both tools that scan dependencies from application binaries, detect the lowest amount of vulnerable dependencies and only reported the known CVEs in them. We manually inspected the tools' result and discuss our observations on their difference in Section 5.1 and 5.2.

Developers may need to know all dependency paths for npm VDs in order to approach fixes: In npm, the same package *A* can be introduced transitively through multiple direct dependencies and therefore, can lie in multiple dependency paths. The developers may need to fix each path separately if there is a vulnerability in package *A*. We find that, npm audit and Snyk reports each dependency path to a VD and considers them as distinct vulnerabilities. For each VD in npm projects, npm audit finds a median of 2 dependency path ($max = 309$ for *lodash:4.6.1* in one project) However, WhiteSource's issues on GitHub only reports one out of many possible dependency paths while OWASP DC also does not report all the possible paths. Dependabot does not report dependency path in its alerts.

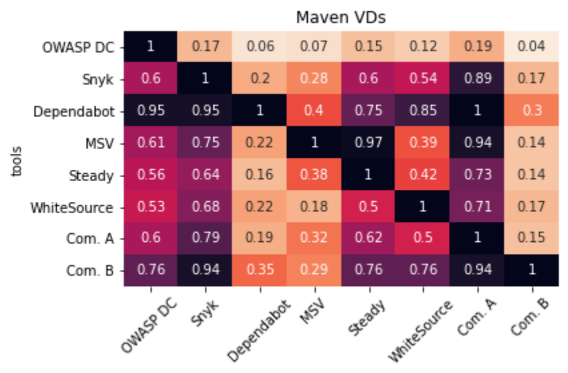
Known vulnerabilities without a CVE identifier are also reported: We find that all tools, except MSV, Commercial B, and C, report non-CVEs as known vulnerabilities. The number of Maven non-CVE vulnerabilities is an average of 6% of the number of CVE vulnerabilities. Overall, the number of npm non-CVE vulnerabilities is an average of the same as of the number of CVE vulnerabilities, though these results are driven by the high number of non-CVE vulnerabilities found by OWASP DC. To understand why the non-CVEs might not have incorporated into the CVE database, we look at their publish date. For the 53 non-CVEs reported by Snyk, 41 were published before 2020; while for 54 non-CVEs reported by WhiteSource, 50 were published before 2020. Therefore, developers may question why a reported vulnerability does not have a CVE identifier as CVE validation usually takes around three months and the scan results are from September 2020. For npm audit, the publish date was usually not present in the results. However, if valid, presence of non-CVEs can be an indicator of the richness of a tool's vulnerability database.

Projects are often scanned in less than a minute: We find tools provide VD reports usually under less than a minute per project. The short scan time can make it possible to integrate VD analysis in continuous integration (CI) tools. GitHub can present its alerts at each code push. Early notifications as such can aid developers from introducing a dependency with already known vulnerabilities. However, additional static and dynamic analysis to assess the risk of the involved vulnerability can take longer time as observed in the case of Steady. A single project, OpenMRS-Core, took 3 hours and 9 minutes to complete the two additional analysis for Steady.

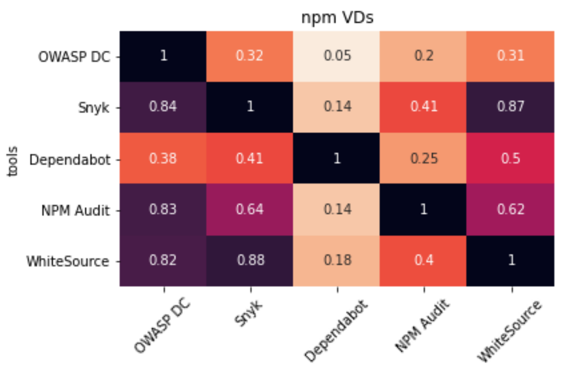
Tools find VDs across all scopes and depths: Table 5 shows a scope breakdown and maximum depth of identified VDs for each tool except Commercial B and C. Commercial B and C scans dependencies during runtime from application binaries consisting of 44 projects, and therefore, we are unable to determine the scope and depth for them. In the table, the same dependency can be listed under different scope for different projects and will be counted twice.

We find that tools detect VDs under all scopes. Also, all tools except Dependabot, scan both direct and transitive dependencies. While maximum depth of identified VD for Dependabot is 2 and 6 for Maven and npm which indicates transitive dependency, we find that these transitive dependencies were identified by Dependabot only due to: a) the Maven dependency file explicitly declared the required version for the transitive dependency; and b) the *lock* file was present in the repository that declared the resolved versions of the full dependency tree. In other cases, Dependabot did not detect transitive VDs.

In subsequent subsections, we discuss how and why the tools' output differed for dependency and vulnerability count. We also discuss the fix actions suggested by the tools.



(a) Overlap ratios for Maven VDs across tools



(b) Overlap ratios for npm VDs across tools

Figure 1: Overlap analysis of unique vulnerable dependencies (VDs) for each tool pair: Cell(i,j) indicates the percentage of i'th tool's VD that are also detected by the j'th tool. For e.g. for maven VDs, 54% of Snyk's detected VDs were also detected by WhiteSource. Conversely, 68% of WhiteSource's detected maven VDs were also detected by Snyk.

5.1 What is the difference among tools' results regarding unique vulnerable dependencies (VDs)?

The heat maps in Figure 1 show overlap ratio across tool pairs for both Maven and npm VDs. For a tool pair (A,B), the heat map shows how many VDs detected by A were also detected by B and vice versa. Note that, Commercial C is not present in the heat map as it was unable to provide a specific version for the identified dependency packages in many cases.

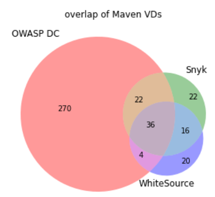
While the count of detected VDs ranges from 17 to 332 for Maven and from 32 to 239 for npm across tools, we find that none of the tools include all the findings from the other tools. Figure 2a, 2b demonstrates the (non-) overlap of detected VDs for three representative tools for both Maven and npm where we see each tool detected some VDs not detected by other tools. In general, Figure 1 shows large overlap between Snyk and Commercial A for maven VDs and between Snyk and WhiteSource for npm VDs.

To understand why the tools' results differ, we manually inspected the results. We specifically focused on the project *coreapps* as this is the project with the largest dependency count and includes both Maven and npm project. While we do not claim to categorize all possible differences, we list below our observations:

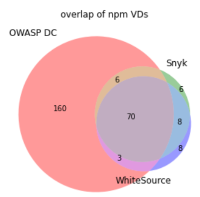
OWASP DC and WhiteSource detected JavaScript dependencies in Maven projects: The Maven projects in OpenMRS can also contain front-end JavaScript files. OWASP DC is able to scan and identify dependencies from JavaScript files such as *jquery*, *handlebars*. These JavaScript dependencies are not resolved by Maven package manager itself. Beside OWASP DC, only WhiteSource detected JavaScript dependencies in Maven projects. In total, 42 JavaScript dependencies were found by OWASP DC while WhiteSource found 20.

Only OWASP DC detected first-party VDs: As mentioned in Section 3.2, OpenMRS can have first-party dependencies – one project listed as a dependency to another project – which were identified only by OWASP DC. Commercial A listed the first-party dependencies as *unmatched libraries*. We also searched through Snyk [16] and WhiteSource's [21] vulnerability database online and found no presence of OpenMRS projects. OWASP DC detected 200 unique VDs that comes from the *openmrs* organization as per package metadata. However, these 200 VDs contain only 14 unique CVEs and 6 non-CVEs with unique description. OpenMRS projects are divided in many sub-modules and OWASP DC reports the same vulnerability separately for each sub-module which results in an inflation of alerts and reported VDs. While OWASP DC is guilty of inflation, the failure of other tools in recognizing vulnerabilities in *openmrs* projects indicates that incompleteness in a tool's vulnerability database can result in false negatives while reporting VDs.

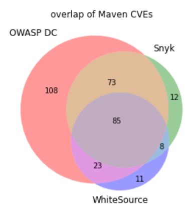
Tools may inflate alerts by repeating the same vulnerability over many related packages: Similar to first-party VDs, we observed tools may report the same vulnerability across many related packages (usually sub-components of a larger package). For example, CVE-2014-3625 was only reported for *spring-webmvc* by MSV, Snyk, Steady, and Commercial A. However, OWASP DC reported this CVE for five separate *spring* packages. The Common Platform Enumeration (CPE) identifier in CVE data maps to *spring_framework* in general and using CPE identifier can result in such inflated results [35]. Similarly, we find that CVE-2014-0114 in *commons-beanutils* is also listed under 3 *struts* packages by OWASP DC and under *struts-core* by Commercial A. *commons-beanutils* is a dependency of *struts* package, and therefore, CVE data reports both the products as affected by the CVE which may be the reason behind such inflated report-



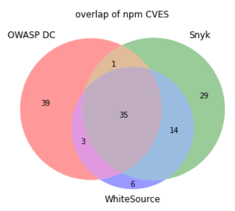
(a)



(b)



(c)



(d)

Figure 2: Venn Diagram for overlap of VDs and CVEs among three representative tools: OWASP DC, Snyk, and WhiteSource. The sub-figures represent overlap of (a) Maven VDs, (b) npm VDs, (c) CVEs in maven VDs, (d) CVEs in npm VDs.

ing. Similarly, OWASP DC reported the same 17 CVEs for *activeio-core*, *activemq-core*, and *kahadb* as they all map to same CPE while other tools only reported *activemq-core*.

Conversely, OWASP DC detected functions of npm packages as individual dependencies. In *lodash* package, OWASP DC detected 31 functions such as *lodash._baseassign*, *lodash._reevaluate* separately besides the package itself, and repeated the same 7 vulnerabilities for each of them.

Tools may have different mapping of vulnerability to dependency: The product identifiers in CVE data can be inaccurate [35] and tools that have self-maintained vulnerability databases can curate vulnerability to dependency mapping for better accuracy. For example, CVE-2014-0114 and CVE-2019-10086 were reported by OWASP DC, WhiteSource, and Commercial A in *commons-beanutils:1.7.0* while MSV reported only CVE-2014-0114 and Dependabot reported only CVE-2019-10086. We looked into Snyk's vulnerability database as to why the CVEs were not reported and found that Snyk lists the affected version range as $[1.8.0, 1.9.2)$ and $[1.9.2, 1.9.4)$ respectively for the two CVEs. Similarly, Dependabot also lists $[1.8.0, 1.9.2)$ version range as affected for CVE-2014-0114 but all versions below 1.9.4 as affected for CVE-2019-10086. In NVD, the affected versions for the two CVEs are listed simply as up to 1.9.1 and up to 1.9.3. Again, CVE-2014-3576 is reported by Snyk in *activemq-core:5.4.3* but not by WhiteSource and Commercial A. We see that Snyk database lists $[0,]$ version range as affected by the CVE while Commercial A lists $5.8.0 - 5.10.1$. We did not find the CVE in WhiteSource vulnerability database. Similarly in npm ecosystem, CVE-2018-1000620 was detected by all tools except Snyk for *cryptiles:0.2.2*. We found that in Snyk's database, the tool lists only $[3.1.0, 3.1.3)||[4.0.0, 4.1.2)$ range as affected by this CVE. The NVD CVE data simply lists version up to 4.1.1 as affected by the CVE.

The presence of *lock* files in npm projects may account for differences in tools' results: As explained in Section 2, npm projects can have *lock* files. Among OpenMRS projects, the *coreapps* project has a *shrinkwrap.json* file, And *idgen* has a *package-lock.json* file. Tools may differ in how they incorporate information from multiple types of dependency files. For example, Dependabot detected 15 VDs from lock files not detected by other tools.

Binary analysis and runtime detection detected low number of VDs, but may be of high priority: We find that Commercial C that analyzed the application binary did not detect any JavaScript dependencies. However, it detected server components such as *jetty*, *tomcat* which are not resolved by Maven package manager. Similarly, Commercial B did not detect any JavaScript libraries. We contacted the tool's customer support, and they informed us that the tool does not cover front-end dependencies (JavaScript). Further, the integration test suite of OpenMRS has only 123 test cases which may not cover the full application and can be a possible reason behind low VDs detected by Commercial B. However, the dependencies found during user interaction may get perceived as high priority by the developers.

Inaccurate mapping of vulnerability to affected versions of packages is a major reason behind alert inflation and difference in tools' results.

5.2 What is the difference among tools' result regarding unique vulnerabilities?

In this section, we discuss the difference among tools' results for both CVEs and non-CVEs.

Apart from inconsistent vulnerability mapping, state of CVEs may also result in differences in tools' results: Figure 2c, 2d demonstrates the (non-) overlap of reported CVEs by different tools. Besides inconsistent vulnerability mapping explained in Section 5.1, there can be other reasons behind CVE differences. CVE-2019-10768 and CVE-2020-7676 were detected by Snyk, Dependabot, and WhiteSource but not by OWASP DC and npm audit. However, the latter two tools reported one of them as non-CVEs with a more elaborate explanation. We observed similar other cases where a CVE was reported without the CVE identifier in npm audit, which therefore, we counted as non-CVE. The other CVE is awaiting reanalysis which may be a possible reason they are not incorporated by the latter tools. Further, we found rejected CVEs to be reported by tools like WhiteSource, Snyk which were not reported by other tools.

Tools report unique non-CVEs not reported by other tools: A comparison between non-CVEs across different tools requires manual analysis as there is no common identifiers. We manually looked at a random sample of VDs that were detected by multiple tools and had unique non-CVEs. For example, for *angular:1.6.1* we observe the following cases: OWASP DC reports two *improper input validation* vulnerability not reported by any other tool. While Snyk, WhiteSource, and Dependabot reported a similar *XSS* vulnerability, Snyk and WhiteSource also reported unique *XSS* not reported by others. Snyk also reported a unique *denial of service* not reported by any other tools. npm audit did not report any of these non-CVEs. We noticed similar differences in non-CVEs for other popular packages, such as *lodash* and *ws*.

5.3 What fix actions are suggested by the tools for generated alerts?

As explained in Section 2, Maven and npm dependencies are resolved and read differently which results in differences in fix approaches for VD. While Maven dependencies are read from the same local cache, npm dependencies create separate copies for a dependency for each possible dependency path.

For Maven projects, tools generally report the versions of the dependency where the corresponding vulnerability has been fixed, if it is fixed. However, mitigating direct and transitive dependencies may require different approaches. Developers can fix the direct dependencies by either removing them or upgrading them to a safer version. For its 4,902 unique alerts on Maven projects, Snyk finds 765 (15.6%) alerts can be fixed through upgrades of direct dependencies. For transitive dependencies, developers either need to upgrade the corresponding direct dependency that pulls a safer version or explicitly declare the version of the transitive dependency. However, both options may introduce a breaking change among the application or its dependencies, especially when a package gets a major version change. Snyk identified 189 unique vulnerabilities that involves 195 dependencies. Out of 195, 40 required a major version change, 37 a minor version change, and the rest required incremental changes.

For npm, the tool can suggest fix options for both direct and transitive dependencies. npm audit treats each dependency path to a VD as a separate vulnerability. For each vulnerability, npm package suggest one out of three resolution options [2]: a) automatic fix: npm audit can perform automatic fixes by adding, removing, upgrading, or moving packages within *node_modules* directory; b) fix involves upgrade to a major release that may break backward compatibility; c) no fix is available, therefore, requires manual review. For *core-apps* project, out of 1,116 alerts, npm audit suggest automatic fix for 84.3% of the alerts; 10.2% required fix involving major, while the rest have no fix available.

Dependabot and WhiteSource can make automatic pull requests on GitHub making updates of a VD to a safer version. WhiteSource (the GitHub pull request bot for WhiteSource is named Renovate) made 94 pull requests and Dependabot made 17 pull requests. Among the pull requests made by the tools, respectively 1 and 5 contained a major version change of a package which may break backward compatibility.

6 RQ2: What additional information is presented by the existing tools to aid in assessing the risk of vulnerability in dependencies?

When reporting a VD, tools report the known vulnerabilities in the dependencies. While all CVEs have a CVSS [3] (Common Vulnerability Scoring System) rating associated with them, most non-CVEs also have a severity rating provided by the tool. However, this severity rating is a characteristic of the vulnerability itself, and the rating is measured from the context of the package containing the specific vulnerability. When a vulnerability lies in a dependency, the risk of the vulnerability may also be determined by how the application uses the dependency – that is – from the context of the dependant application We notice that, no VD detection tool reports any risk rating of a vulnerability in dependency from the context of the application itself.

However, the studied VD detection tools reported several additional metrics in their scan reports. We identified the metrics that may *possibly* aid developers in assessing the risk of the vulnerability in their project's dependency and characterized them in five categories, as discussed in the next five subsections:

6.1 Code analysis based metrics

Tools may analyze source code or binary of an application to infer which dependencies are in use; and if vulnerable part of the dependency code is (potentially) executed by the application. Three of the tools, Steady, Commercial A, and B make use of below code analysis based metrics:

Reachability Analysis: Tool can curate their vulnerability database with details on which part of the code (e.g. method, class) is involved in a specific vulnerability. Tools then can infer if the vulnerable code is potentially reachable from the dependant application through static and/or dynamic analysis.

Steady constructs static call graphs of an application, and if the vulnerable code in the dependency is reached through the call graph, Steady marks the corresponding alert as *potentially executable*. Further, Steady can look at the executability traces through unit testing to determine if the vulnerable code is *actually executed*. Commercial A, similarly, can perform static analysis to provide developers with vulnerable call chain – that is – the call chain from the application code that reaches the vulnerable method of the dependency.

Steady: Static Analysis (Vulnerable code potentially executable)			
Total Alerts	Package not in use	Non-vulnerable code of package used	Vulnerable code of package used
2,489	2,095 (84.2%)	340 (13.7%)	54 (2.1%)
Steady: Dynamic Analysis (Code actually executed)			
Total Alerts	Package not in use	Non-vulnerable code of package used	Vulnerable code of package used
2,489	2,437 (97.9%)	11 (0.4%)	41 (1.6%)
Commercial A: Vulnerable call chains			
Total Alerts	Vulnerable Method Calls	Total Vulnerable Call Chain	Median Call Chain per Method
2,998	31	93	2.0

Table 6: Code analysis based prioritization metrics: Vulnerable code reachability analysis

Table 6 shows the reachability analysis from Steady and Commercial A. We find that for 84.2% of the alerts, Steady did not find the corresponding dependency to be used at all by the dependant application. For only 2.1% alerts, Steady found the vulnerable code of the dependency was *potentially executable* through static analysis. For 1.6% of the alerts, Steady found the vulnerable code in the dependency was *actually executed* by the dependant application through dynamic analysis. However, we find disconnect between the findings of static and dynamic analysis. Only for 13 alerts, both static and dynamic analysis found the vulnerable code to be in use. Also, for 11 alerts where dynamic analysis have found the vulnerable code to be actually executed, static analysis did not find any part of the dependency code to be in use. This observation may indicate limitations to reachability analysis. Similar to Steady, Commercial A also found a low number of cases where the vulnerable code of the dependency can actually be reached from application source code.

Static analysis, such as call graph construction for Java, is known to have limitations [49]. The effectiveness of dynamic analysis such as Steady's, is also dependant on having a good test-suite and test coverage. Further, OpenMRS uses mocking for testing in many cases, which the Steady skips. Additionally, we set Steady to skip whenever faced with a test case failure. We see that OpenMRS projects reach only around 20% test coverage in Steady. The limited test coverage may have affected the dynamic analysis findings for Steady. Nevertheless, the additional information on reachability for certain alerts may aid developers in assessing the risk and how the corresponding vulnerability can be exploited from the context of their application.

Dependency Usage: The dependant application may only use a portion of the functionalities offered by a dependency. How much functionalities of a dependency is used by an application may indicate the probability of the vulnerable code in the dependency getting used by the dependant application.

Steady, through its static and dynamic analysis, reports how many classes out of total available is used in a dependency. Similar metric is reported by Commercial B but through interaction with the application. For example, based on integration testing, Commercial B found 203 out of 414 classes (49%) for *spring-web* and 790 out of 4,4414 (17.9%) classes for *groovy-all* to have been used by OpenMRS.

6.2 Package Based metric:

The characteristics of the package itself that is being used as a dependency may indicate the risk associated with it. For example, a package that is not actively maintained or have poor code quality, may contain high risk vulnerability.

Package security rating: Commercial B provides a letter grade on how secure it is to use a package as a dependency. Out of the 17 packages being identified as VD by Commercial B, 16 have an *F* rating while one has a *D* rating. The tool calculates the security rating of a package based on its age, released versions, and number of known vulnerabilities.

6.3 Dependency characteristics based metrics

The scope and depth of the dependency may indicate the risk of the vulnerability it contains.

Dependency scope: We observed that for npm projects, the dependency scope is usually mentioned in the tools' result. Snyk does not report the *dev* dependencies by default while npm audit can also filter results based on scope. However, for Maven projects, only Steady reported the scope for each VD. The dependencies that are only used during testing phase may perceived with lower risk by the developers.

Dependency depth: Risk may be associated with how deep a dependency lies within the dependency tree. Snyk and Steady mentions if a dependency is direct or transitive for Maven projects. For npm, Snyk and npm audit reports all possible dependency paths for each VD.

6.4 Vulnerability based metric:

The characteristics of the vulnerability itself can be used in assessing risk. We found below three types of information provided by tools:

Severity: Tools typically report the CVSS scores for the CVEs. For the non-CVEs, Snyk and Commercial A also present a CVSS score. However, GitHub Dependabot and npm audit presents severity rating on a scale of their own for both CVEs and non-CVEs. For both the tools, the scale consists of four levels similar to CVSS3 levels: *low*, *moderate*, *high*, and *critical*. We compared these two tools' rating with CVSS3 scores for the CVEs they reported and found that both tools generally has a higher rating than CVSS3 for the same CVEs. Out of 30 CVEs that npm audit detected, it had higher rating than CVSS3 for 15 (50%). Similarly, out of 86 CVEs that Dependabot detected, it had higher rating for 54 (67.5%)

Available exploits: The availability of known exploits may contribute in assessing the risk for a vulnerability in a dependency. For each vulnerability, Snyk provides if an exploit is publicly available. For 310 Snyk vulnerabilities, Snyk reports 218 do not have a public exploit; 10 have a functional exploit; 37 have a proof of concept exploit; while 45 have unproven exploits available. Beside Snyk, Commercial A also reports on available exploits.

Popularity: How popular or well-known is a vulnerability may indicate the probability it may get exploited in the wild. Steady integrates Google trend analysis for each vulnerability in its reports which indicates the search hits within past 30 days for the specific vulnerability.

6.5 Confidence in alert validity:

As security tools may generate false positive alerts, a confidence rating for each alert may aid in developers in prioritizing auditing.

Evidence count: As OWASP DC detects dependencies from scanning multiple sources, it can provide the quantity of data extracted as a proof for each dependency. The tool provides confidence label, from *Low* to *Highest*, based on this evidence count. For all Maven alerts except 4, OWASP DC provided with either *High* or *Highest* confidence rating. Similarly, all npm alerts were labeled with *Highest* confidence.

7 Discussion

Tools may need to scan source code and binaries as well besides dependency file to find all the dependencies and to infer which of them are in use. Dependency files are the typical source to resolve dependencies for a project. However, we find that OWASP DC can find JavaScript front-end components as well when scanning source code files which indicates that there can be code components in a project, possibly in different programming language, not mentioned in its dependency files. Tools can miss such dependencies when solely relying on dependency files. Further, static analysis on source code may help to determine which dependencies are in use. Similarly, dynamic analysis and runtime detection can provide more evidence on executable code in dependency and therefore, assess the risk of specific vulnerabilities.

Accurate vulnerability to dependency mapping should be ensured by the tools in order to avoid both false positive and false negatives: We showed examples in Section 5.1 on how inconsistency in vulnerability to dependency mapping can result in differences between tools' results. The product identifier (CPE) provided in CVE data may not be suitable for vulnerability mapping for VD detection tools as we see evidence of OWASP DC inflating alerts by repeating the same vulnerability for several related packages having the same CPE identifier. We also find inconsistency in what version range is listed as affected for a specific CVE by different tools. Our findings highlights the importance of a more accurate mapping in public vulnerability databases for VD detection.

Non-CVEs should get reported to CVE database to prove their validity and cross-tool vulnerability mapping. We find VD detection tools report known vulnerabilities in dependencies that do not have a CVE identifier, even though many of them were published before 2020 leaving the non-CVEs with enough time to be incorporated into CVE database. We also see tools report rejected CVEs and CVEs awaiting reanalysis which sheds light on the possible lack of validity of reported vulnerabilities. We find that different tools can report the same non-CVEs. However, cross-tool referencing cannot be done automatically without a common identifier like CVE. We suggest reporting the non-CVEs to CVE database to establish validity, make cross-tool referencing easy, and make the vulnerability widely known.

Tools should suggest fix options while explaining the risk of backward compatibility. We find that npm audit suggests fix option for each vulnerable dependency path. However, for Maven project, the fix of the transitive dependencies may require more manual intervention to analyze the potential risk of a version change. Prior work has found that fear of breaking change is one of the primary reasons developers do not want to update VDs [22]. Tools may provide more in-depth analysis on what code change is there with a certain version change in a dependency and how it may affect the dependant application.

Reachability analysis can help developers identify how the vulnerability in a dependency is relevant to their application. We find that two studied tools, Steady and Commercial A, provide static and dynamic analysis in order to determine if an application can potentially execute the vulnerable code part in the dependency. However, such analysis requires a vulnerability database enriched with details, such as code constructs, involved in the vulnerability. Beside Steady and Commercial A, we find that Snyk also provides details on which code constructs are involved in the vulnerability and their corresponding fixes. While reachability analysis can provide developers with evidence on how their application reaches a specific vulnerability, the disconnect between Steady's static and dynamic analysis findings, as shown in 6.1, indicates that more research is necessary on such analyses.

Future research is required to evaluate metrics that can aid in assessing the risk of a VD. We find that no tools provide any risk rating of a VD from the context of the dependant application. Prior work has found that many vulnerabilities in the dependency may not be relevant to the application itself [53]. We have seen tools provide code analysis based metrics determining if a dependency is used in the production environment or not and a reachability analysis for the vulnerable code. However, the effectiveness of such analysis needs to be evaluated and how they can be improved by incorporating additional analyses, e.g. how close is a dependency to the application's attack surface [50]. Further, we have seen tools to provide non-code based metrics such as package security rating, vulnerability exploits and popularity. Multiples VDs can be under use in runtime and developers may need to prioritize their fixes given the fixes are not cheap [46]. Some areas of future work include how do practitioners prioritize VDs; what metrics they use in the decision making process; and how effective are those metrics.

8 Threats to Validity

The threats to internal validity of our study involve the selection of the evaluation subject. We explain our rationale behind the choice of OpenMRS in Section 3.1. The threats to external validity of our study involves the selection of the studied tools. While we are unable to cover all existing VD detection tools across different programming languages, we do not claim the observations we have in Section 5.1, 5.2, 5.3 and the characterizations we provide in 6 to be exhaustive.

9 Related Work

Dependency network of package ecosystems and the presence of known vulnerabilities in dependencies have been studied in the literature [28, 34]. Decan et al. [27] studied the impact of security vulnerabilities in npm dependency network. and found that the number of packages with a known security vulnerability is growing over time, and half of the dependant packages do not get fixed even when the fix is available for the upstream dependency. [31] and [37] also found around one-third of the packages in npm network to have at least one vulnerable dependency (VD) while [31] found that *context use of the module* and *breaking changes* are potential reasons for not resolving the VDs.

The potential impact of the vulnerabilities in dependency has also been studied. Zapata et al. [53] studied the impact of a vulnerability in *ws* package in npm network on applications that were using the vulnerable version of the package. The study finds 73.3% of the dependant applications did not use the vulnerable code. The study also finds that the dependant applications that do not use the vulnerable code take longer to migrate to new versions of a dependency. To detect VDs where the vulnerable code is actually used by the including application, Ponta et al. [42–44] proposed a *code-centric* and *usage-based* approach based on which the tool Steady was developed. Paschenko et al. [40] discusses the over-inflation problem when reporting VDs. with unexploitable vulnerabilities. In another work, Paschenko et al. [41] interviewed developers on dependency detection tools. The study found that developers think dependency detection tools generate many *irrelevant* and *low-priority* alerts and may even rely on social channels than dependency detection tools for vulnerability reporting. Developers recommended dependency detection tools to *report only relevant alerts, work offline, and be easily integrated into company workflow*. The literature focuses on the importance of assessing the risk of vulnerabilities in a dependency with respect to the dependant application so that developers can prioritize accordingly.

To the best of our knowledge, there has been no formal study yet evaluating and comparing the existing VD detection tools. The closest to our work is a recent study by Ponta et al. [44] where the authors compared their research tool Steady with OWASP Dependency-Check. The study compare the two tools over a sample of the alerts generated on Java applications. The comparison was performed from the perspective of rechability of a vulnerability in the dependency. The study finds both tools to have their unique findings. The study also finds Steady has no false positives but a few false negatives while OWASP DC has a non-negligible false positives. However, the study focuses on evaluating the detection capabilities of Steady based on its advanced analysis whereas our study aims to provide a comprehensive comparison of existing VD detection tools for both Java and JavaScript dependencies.

10 Conclusion

We evaluate 10 VD detection tools on a large web application composed of Maven (Java) and npm (JavaScript) projects. We find disconnect between the tools' results while no single tool covered all the findings from the other tools. We find results by the tools can both be inflated based on how the data is presented (e.g. repeating the same vulnerability across several related packages) and deflated based on the scanning technique (e.g. interaction testing). Evidences in our findings suggest that accurate vulnerability to dependency (package:version) mapping is required to avoid both false positives and false negatives. Further, we find that tools can provide additional metrics to assess the risk of a VD from the context of the dependant application. We characterize the provided metrics into five high-level categories. Overall, we find that tools should strive to maintain an accurate vulnerability database while future research is necessary in evaluating the metrics for assessing the risk of vulnerabilities in dependency.

11 Acknowledgements

We thank the [blinded] for their valuable feedback. Our research was partially funded by NSA.

References

[1] About semantic versioning. https://docs.npmjs.com/about-semantic-versioning.

[2] Auditing package dependencies for security vulnerabilities. https://docs.npmjs.com/auditing-package-dependencies-for-\security-vulnerabilities.

[3] Common vulnerability scoring system. https://en.wikipedia.org/wiki/Common_Vulnerability_Scoring_System.

[4] Eclipse steady 3.1.14 (incubator project). https://eclipse.github.io/steady/about/.

[5] Evaluation framework for dependency analysis. https://github.com/srcclr/efda.

[6] Github advisory database. https://github.com/advisories.

[7] How does dependency-check work? https://jeremylong.github.io/DependencyCheck/general/internals.html.

[8] Mitre cve datgbase. https://cve.mitre.org/.

[9] Module counts. modulecounts.com.

[10] National vulnerability database. https://nvd.nist.gov/vuln.

[11] Npm security advisories. https://www.npmjs.com/advisories.

[12] Openmrs around the world. http://guide.openmrs.org/en/.

[13] Openmrs reference application distribution. https://wiki.openmrs.org/display/docs/OpenMRS+Reference+Application+Distribution.

[14] Openmrs sdk. https://wiki.openmrs.org/display/docs/OpenMRS+SDK.

[15] Snyk open source security management. https://snyk.io/product/open-source-security-management/.

[16] Snyk vulnerability db. https://snyk.io/vuln.

[17] Sonatype oss index. https://ossindex.sonatype.org/.

[18] Steady vulnerability dataset. https://github.com/SAP/project-kb.

[19] Victims software vulnerability scanner. https://blog.victi.ms/.

[20] Whitesource bolt for github. https://github.com/apps/whitesource-bolt-for-github.

[21] Whitesource vulnerability database. https://www.whitesourcesoftware.com/vulnerability-database/.

[22] 0patch.com. Security patching is hard. https://0patch.com/files/SecurityPatchingIsHard_2017.pdf.

[23] 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.

[24] Yang Chen, Andrew E Santosa, Ang Ming Yi, Abhishek Sharma, Asankhaya Sharma, and David Lo. A machine learning approach for vulnerability curation. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 32–42, 2020.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Josh Fruhlinger. Equifax data breach faq: What happened, who was affected, what was the impact? https://www.csoonline.com/article/3444488/equifax-data-breach-faq-what-happened-who-\was-affected-what-was-the-impact.html, 2020.

[31] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? 2015.

[32] 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.

[33] 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.

[34] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017.

[35] Sean Kinzer. Using cpes for open-source vulnerabilities? think again. https://www.veracode.com/blog/managing-appsec/using-cpes-open-source-vulnerabilities-think-again.

[36] 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.

[37] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018.

[38] National Institute of Standards and Technoloy (NIST). Guide for conducting risk assessments, nist special publication 800-30. https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final, September 2012. [Online; accessed 7-Oct-2020].

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

[40] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 2020.

[41] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. *Proc. of CCS*, 20.

[42] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–420. IEEE, 2015.

[43] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018.

[44] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, pages 1–41, 2020.

[45] Apache Maven Project. "introduction to the dependency mechanism". http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[46] Teri Radichel. Why patching software is hard: Technical challenges. https://www.darkreading.com/vulnerabilities-and-threats/why-patching-software-is-hard-technical-\challenges-/a/d-id/1330181.

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

[48] Help Net Security. Surge in cyber attacks targeting open source software projects. https://www.helpnetsecurity.com/2020/08/13/surge-in-cyber-attacks-targeting-open-\source-software-projects/?utm_source=feedburner&utm_medium=feed&\utm_campaign=Feed%3A+HelpNetSecurity+%28Help+Net+Security%29.

[49] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. ICSE, 2020.

[50] Christopher Theisen, Nuthan Munaiah, Mahran Al-Zyoud, Jeffrey C Carver, Andrew Meneely, and Laurie Williams. Attack surface definitions: A systematic literature review. *Information and Software Technology*, 104:94–103, 2018.

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

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

[53] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563. IEEE, 2018.

[54] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919, 2017.

[55] Zeljka Zorz. The percentage of open source code in proprietary apps is rising. https://www.helpnetsecurity.com/2018/05/22/open-source-code-security-risk/.