A comparative study of vulnerability reporting by software composition analysis tools

Nasif Imtiaz
simtiaz@ncsu.edu
North Carolina State University
North Carolina, USA

Seaver Thorn
swthorn@ncsu.edu
North Carolina State University
North Carolina, USA

Laurie Williams
lawilli3@ncsu.edu
North Carolina State University
North Carolina, USA

ABSTRACT

Background: Modern software uses many third-party libraries and frameworks as *dependencies*. Known vulnerabilities in these dependencies are a potential security risk. Software composition analysis (SCA) tools, therefore, are being increasingly adopted by practitioners to keep track of vulnerable dependencies. *Aim:* The goal of this study is to understand the difference in vulnerability reporting by various SCA tools. Understanding if and how existing SCA tools differ in their analysis may help security practitioners to choose the right tooling and identify future research needs. *Method:* We present an in-depth case study by comparing the analysis reports of 9 industry-leading SCA tools on a large web application, OpenMRS, composed of Maven (Java) and npm (JavaScript) projects. *Results:* We find that the tools vary in their vulnerability reporting. The count of reported vulnerable dependencies ranges from 17 to 332 for Maven and from 32 to 239 for npm projects across the studied tools. Similarly, the count of unique known vulnerabilities reported by the tools ranges from 36 to 313 for Maven and from 45 to 234 for npm projects. Our manual analysis of the tools' results suggest that accuracy of the vulnerability database is a key differentiator for SCA tools. *Conclusion:* We recommend that practitioners should not rely on any single tool at the present, as that can result in missing known vulnerabilities. We point out two research directions in the SCA space: i) establishing frameworks and metrics to identify false positives for dependency vulnerabilities; and ii) building automation technologies for continuous monitoring of vulnerability data from open source package ecosystems.

KEYWORDS

software composition analysis, supply chain security, vulnerability, dependency, security tools, case study

1 INTRODUCTION

Most modern software uses third-party open source libraries, packages, or frameworks that are referred to as *dependencies*. A Black Duck report found 98% of the 1,546 audited commercial codebases in 2020 contained open source packages with an average of 528 packages per codebase [47]. However, known vulnerabilities in dependencies are one of the top ten security risks [37]. The Black Duck audit also found 84% of the codebases to contain at least one publicly known vulnerability in their open source dependencies [47].

Software composition analysis (SCA) tools are used to report known vulnerabilities in the open source dependencies of a software. However, these tools may differ in how they detect the dependencies and the vulnerability database they maintain. A comparative study is yet to be performed to review the existing SCA tools and determine if and how they differ. Furthermore, not all alerts generated by SCA tools are relevant or high priority to the developers [39]. If and how existing SCA tools aid developers in assessing the risk of the vulnerabilities from the context of the client application needs to be studied to help future research.

The goal of this study is to aid security practitioners and researchers in understanding the vulnerability reporting by software composition analysis tools through a comparative study of these tools on a real-world case study. Our research questions are:

> **RQ1:** What are the differences between vulnerability reports produced by the different software composition analysis (SCA) tools?
>
> **RQ2:** What metrics are presented by the SCA tools to aid in the risk assessment of dependency vulnerabilities?

To answer, we present an in-depth case study by running 9 SCA tools on a large web application, OpenMRS, that utilizes two popular package ecosystems. The application consists of 43 Maven (Java) and 5 npm (JavaScript) projects. The studied SCA tools vary in their scanning technique and vulnerability database, and represent the state-of-the-art. The contributions of this paper include the first evaluation of the SCA tools through (a) a quantitative comparison of their vulnerability reports on a real-world case study, (b) a manual analysis of the differences among the tools' reports, and (c) characterization of metrics provided by the tools for assessment of the dependency vulnerabilities.

The remainder of the paper is structured as follows: Section 2 introduces the key concepts and terminologies; Section 3 and 4 explains the evaluation case study and the studied SCA tools. Section 5 discusses the findings of this paper, followed by discussion and limitations of the findings. Section 8 discusses related work, followed by conclusion.

2 KEY CONCEPTS & TERMINOLOGIES:

Dependency: When a software uses an open source package, the package is referred to as *dependency* of the software. Typically, a software declares a specific or a range of valid versions of a package as its dependency in a manifest file that we refer to as *dependency file*. However, a software may use open source package or code fragments without explicit declaration as well [29]. In the remainder of this paper, we refer to 'dependency' as a specific version of a package. For example, version 1.0.0 and version 2.0.0 of the same package A will be considered as distinct *dependencies*. However, they will be considered as the same *package*.

The dependencies declared through dependency files are resolved through some *package manager*. *pom.xml* and *package.json* are dependency files for Maven and npm package manager, respectively. 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 that are 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. The *depth* of a dependency refers to their level in the dependency tree, with direct dependencies having a depth of one.

Vulnerability: NIST [36] defines vulnerability as "weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source." If a vulnerability gets exploited by a threat source, the potential for loss or damage is referred to as *risk* of the vulnerability. Vulnerabilities can get discovered in already released versions of software packages. If reported, respective package maintainers can fix the vulnerability in a new version. When a dependency of a software is subject to publicly known vulnerabilities, it is referred to as a *vulnerable dependency*.

Software Composition Analysis (SCA): SCA is a part of application analysis that deals with managing open source use. SCA tools typically generate an inventory of all the open source components in a software product and analyze the license compliance and the presence of any known vulnerabilities in them. By the vulnerability detection capability of SCA tools, we mean the ability to identify and report known vulnerabilities in the open source components used by a software application.

Disclosed and Discovered Vulnerabilities: The National Vulnerability Database (NVD) [8] is the U.S. government repository of publicly accessible standards-based vulnerability management data. The primary reference-tracking system for publicly disclosed vulnerabilities in the NVD database is the Common Vulnerabilities and Exposure (CVE) system where each vulnerability is referenced by a unique **CVE** identifier, a system developed by Mitre [7].

Additionally, SCA tools augment NVD vulnerabilities/CVEs with vulnerabilities found in other databases, such as npm Security Advisories [9], Sonatype OSS Index [16], and GitHub security advisories [5], that do not necessarily have a CVE identifier. Similarly, SCA tools can also have proprietary techniques to discover vulnerabilities in open source packages [23, 50] as explained in Section 4.1. In this paper, vulnerabilities reported by SCA tools that do not have an associated CVE identifier are referred to as *Non-CVEs*.

2.1 Maven:

Maven is a package manager for Java projects.

Dependency Scopes: Maven dependencies can have six different scopes [43]: compile, provided, runtime, test, system, and import. The scopes determine the phase when a dependency will be used 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 that is read from a local repository. In the dependency file (*pom.xml*), developers generally specify a single version for its dependencies. 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.

Dependency Mediation: npm copies all the dependencies in a project sub-directory called 'node_modules', with a similar structure of the dependency tree. If two dependencies A and B both depend 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 the *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. npm packages follow the SemVer format [1] for version numbering.

3 EVALUATION CASE STUDY: OPENMRS

OpenMRS is a web application for electronic medical record platform [11]. 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" [12]. We choose Version 2.10.0 of this distribution released on April 6, 2020 (the 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. Out of the 44 projects, 39 are Maven projects and 1 is a npm project. The other 4 projects are composed of a Maven and a npm project each. Based on OpenMRS structure, **we scope our study to Maven and npm dependencies.** We use OpenMRS SDK [13] to automate the build, test, and run of the individual projects and assemble the full application in this study.

3.1 Why OpenMRS?

Choosing test cases to evaluate software security tools can be a complex task. For comparison of security tools, Delaitre et al. [28]

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

notes that the test case should have *sufficient and diverse number* of security weaknesses. OpenMRS depends on many third-party dependencies as will be seen in Section 3.2; and being a web application, is composed of several heterogeneous components, such as database, content generation engines, client-side code etc., therefore increasing the probability of having a large, diverse set of vulnerable dependencies.

Another approach of comparison instead of a single case study can be running the tools on a group of diverse projects. However, three of the selected tools in this study (Steady, Commercial A, and B) are (a) resource and time-consuming to set up and run; (b) involve certain requirements, e.g., acceptance tests for interactive binary instrumentation, unit tests for executability tracing; and (c) involve permission issues in case of the commercial tools. On the contrary, focusing on a single case study enables us to manually investigate the differences in the tools' results.

OpenMRS has also been used in security research in the past [22, 24, 25, 34, 45, 48]. [34] evaluated OpenMRS for medical system security requirements; [45] evaluated OpenMRS for access control checking; while [22] 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 `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 *internal* dependencies – that is – a project within the OpenMRS distribution can be listed as a dependency for another project. We do not count the internal dependencies in Table 1. Also, npm projects can contain *lock* files such as *shrinkwrap.json*, *package-lock.json* which are not considered.

4 SCA TOOLS

In this section, we explain the criteria we use to select the SCA tools; description of the tools; how we performed the scan on OpenMRS; and how we analyzed the reports produced by the tools.

4.1 Selection Criteria

To identify the existing SCA tools from both industrial offerings and the latest research, we performed an academic literature search and a web search through the 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 the following *inclusion criteria*: a) scans either Maven or npm projects; b) we have access to an executable tool; and c) offers unique features when compared with already selected tools. From our selection process, we selected nine tools. Two of the tools are not freely available, and the license agreements prevent us from providing names. We refer to them as Commercial A and B. **Out of the selected 9 tools, 4 tools can scan both Maven and npm projects, 1 tool scans only npm projects, while 4 tools scan only Maven projects.**

We observed that SCA tools primarily differ in three dimensions:

(1) **Vulnerability database:** To report the list of known vulnerabilities, the tools need a database. Tools can pull vulnerability data from *third-party* source(s) such as NVD CVEs [8]. Additionally, SCA tools can maintain their own vulnerability database where they collect and verify vulnerability data through different techniques [23, 50].

(2) **Dependency scanning source:** SCA tools can detect open source dependencies from dependency manifest file, source code, and binaries. Typically, dependency files are the common source to resolve dependencies of a project as is done by the package managers as well.

(3) **Additional analysis to infer dependency use:** Tools can perform additional static and/or dynamic analysis to infer how the dependencies are being used by an application.

4.2 Tool description

For the selected tools, we describe (a) if they scan Maven or npm dependencies, their (b) data source, (c) scanning technique, and (d) how we performed the scan for this study.

OWASP Dependency-Check (DC): This tool scans both Maven and npm projects and works by scanning the dependency files, JARs, and JavaScript files [6]. It pulls vulnerability data from multiple third-party sources including NVD, OSS Index, npm advisories. 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 to perform JavaScript scanning.

Snyk: Snyk also scans both Maven and npm projects. The tool works by scanning dependency files [14] and maintain its own vulnerability database [15]. We ran the command line tool (Version 1.382.0) that is freely available through the command *snyk test –all-projects –dev –json*.

GitHub Dependabot: Dependabot scans both Maven and npm projects hosted on GitHub. GitHub maintains its own vulnerability

database [5] where it pulls data from NVD, npm advisories. Additionally, maintainers on GitHub can publish vulnerabilities in their projects as well. We hosted the 44 studied projects on the first author's GitHub account and retrieved the Dependabot alerts through GitHub API.

Maven Security Versions (MSV): This tool only scans Maven projects [18] through dependency files. We ran this tool through its Maven plugin.

npm audit: This is a native tool of npm package manager for scanning npm projects. The tool works by scanning dependency files and maintains its own vulnerability database [10]. We used the `npm audit –json` command.

Eclipse Steady: This tool only scans Java (Maven) projects. The tool performs additional analysis to assess the execution of vulnerable code in the dependencies of an application [4]. The approach implemented is described in [41] and [40]. The tool requires a manual set up, along with the vulnerability database provided by the tool. We used Version 3.1.10 of this tool. We set up Steady in a virtual machine, allocating 16 GB RAM, and 4 processor cores. Steady hosts their vulnerability data set on GitHub [17]. The data set contains patch commit information for each vulnerability. We imported the data source updated on Jan 24, 2020. We then performed the patch analysis feature provided by the tool to identify the involved code constructs for each vulnerability. For reachability analysis of the identified vulnerabilities, Steady performs three analyses: 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 bot named "WhiteSource Bolt" [19] which scans both Maven and npm projects. WhiteSource also maintains its own vulnerability database [20]. We connected the GitHub bot 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 to 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 manual intervention. This tool offers static analysis by default and dynamic analysis as an option to identify vulnerable call chains. We received results only with static analysis performed on the code. The tool maintains its own vulnerability database.

Commercial B: We used the free cloud edition of the tool, where it only scanned the Java dependencies (the customer support informed us that the tool does not scan front-end libraries). The tool checks for the reachability of vulnerabilities in 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 testers. The tool uses third-party vulnerability databases including NVD, which they curate themselves to enhance accuracy. To run this tool on OpenMRS, we make use of 123 test cases provided by OpenMRS for integration testing that interact with the application through a Selenium webdriver. We connected OpenMRS to this tool and used the integration test suite to interact with the application.

We collected the scan reports separately for 44 projects for 8 tools. For Commercial B, which analyzes the application during runtime, we get a single report for the whole OpenMRS distribution. As vulnerability data gets updated over time, we ran all the tools during September 2020 to ensure a fair comparison, except for Steady whose vulnerability data is from January 2020.

4.3 Analyzing Tool Results

Below, we discuss the 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. We observed that the same alerts can be repeated in tools' reports for various reasons. The alert count, however, may indicate the amount of audit effort required from the developers.

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 the vulnerability description along with the affected package(s) are 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 of 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. Commercial B monitors dependency during runtime through interaction, therefore, also does not have a definite scan time.

Other information: Tools had additional information in their reports, 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 when discussing the findings.

Manual analysis of the tools' report: To understand why there are differences in the tools' results, we manually inspected the tools' results. We specifically focused on the project *coreapps* as this is the project with the largest dependency count and includes both Maven and npm dependencies. The first author went through results from all the tools for *coreapps*, and categorized the differences. The second author then independently went through the results from the studied tools and verified the categorization done by the first author.
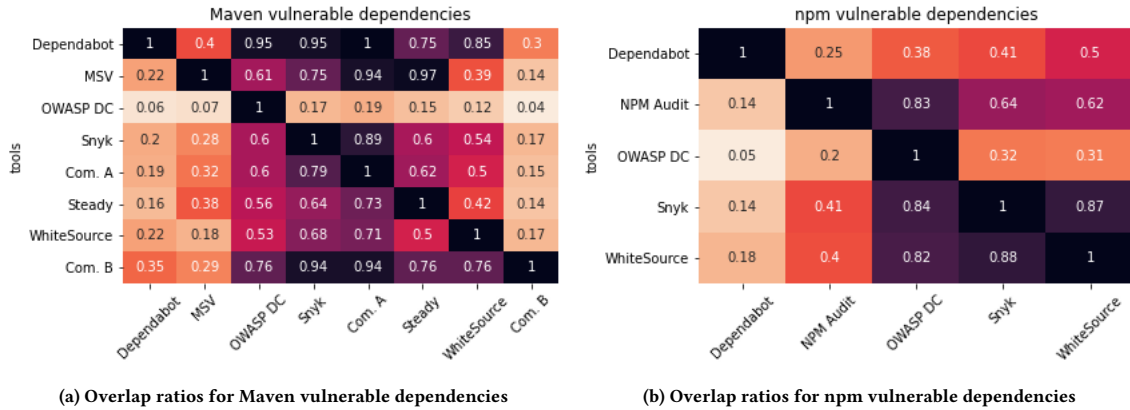
5 FINDINGS

In this section, we present descriptive statistics on how SCA tools differed on vulnerability detection, a manual analysis on why the tools differed (RQ1); and a characterization of the metrics provided by the studied tools for aiding in risk assessment of vulnerability in dependencies, (RQ2).

Table 2: 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

Table 3: 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

(a) Overlap ratios for Maven vulnerable dependencies

(b) Overlap ratios for npm vulnerable dependencies

Figure 1: Overlap analysis of unique vulnerable dependencies for each tool pair: Cell(i, j) indicates the percentage of i'th tool's reported vulnerable dependencies that are also reported by the j'th tool.

5.1 RQ1: What are the differences between vulnerability reports produced by the different software composition analysis (SCA) tools?

Table 2 and 3 show the tools' result summary for Maven and npm dependencies, respectively. The table provides the total count for alerts and unique dependencies, dependency paths (for JavaScript), packages, and vulnerabilities for OpenMRS as a single application. The tables also report the total count of CVEs, non-CVEs, and scan time. For the eight tools that scanned projects individually (besides Commercial B), the tables provide in parentheses the median count per project for alerts and for unique dependencies, dependency paths, packages, and vulnerabilities.

The alert counts are higher than the count of unique vulnerabilities or dependency paths, as discussed in Section 4.3. While the total alert count repeats the same vulnerabilities found across projects, some tools repeat the same alert within a project as well

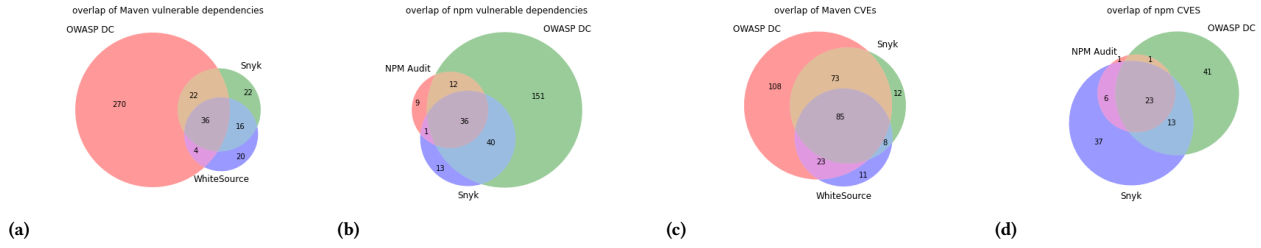**(a)**　　　　　　　　**(b)**　　　　　　　　**(c)**　　　　　　　　**(d)**

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

Table 4: Scope breakdown, rate of direct dependencies among reported vulnerable dependencies (VDs), and max. depth for the reported transitive VDs

Tool	Scope breakdown for Maven VDs				Scope breakdown for npm VDs		Direct VDs (across all projects)		Max. Depth of VDs	
	Compile	Provided	Runtime	Test	Prod	Dev	Maven	npm	Maven	npm
OWASP DC	58	66	4	54	65	207	5.8%	4.4%	6	10
Snyk	56	62	2	25	13	83	14.8%	3.5%	7	10
Dependabot	15	5	1	2	6	8	97.0%	65.9%	2	6
MSV	19	30	1	3	NA	NA	1.7%	NA	5	NA
npm audit	NA	NA	NA	NA	15	51	NA	0.9%	NA	10
Steady	60	60	4	11	NA	NA	5.3%	NA	5	NA
WhiteSource	54	0	2	0	12	76	60.0%	8.8%	5	10
Commercial A	72	79	1	0	NA	NA	8.7%	NA	5	NA

due to modular project structure. 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, while npm can have multiple versions of the same package as dependencies even within a single project. We now discuss how the SCA tools have differed in their reporting:

The tools differed both on identifying unique vulnerable dependencies and the unique vulnerabilities: OWASP DC detects the highest number of unique dependencies and unique vulnerabilities for both Maven and npm projects. However, our analysis in Section 5.2 indicates more may not necessarily be better. Conversely, Commercial B that monitors the dependency under use during runtime detected the lowest number of vulnerable dependencies for Maven projects. MSV and Dependabot detected the lowest number of unique vulnerabilities, respectively, for Maven and npm projects.

5 out of the 8 tools for Maven reported non-CVEs while all the 5 tools for npm reported non-CVEs. We observe that npm packages have a higher proportion of non-CVEs to CVEs than Maven packages. We find OWASP DC to report higher non-CVEs than any of the other 4 tools for npm projects. However, as OWASP DC does not provide an identifier for non-CVEs, we tracked unique non-CVEs through vulnerability description and affected packages, which may have resulted in duplication of the same vulnerabilities.

Only 2 out of the 5 tools that scanned npm projects report vulnerable dependency paths: 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 only npm audit and Snyk report all possible dependency paths to each unique vulnerability.

Tools have non-overlap in reported vulnerabilities and dependencies: We measured how much of the unique vulnerable dependencies reported by the tools overlap with each other. The heat maps in Figure 1 show overlap ratio across tool pairs for both Maven and npm projects. For a tool pair (*A,B*), the heat map shows how many dependencies reported by *A* were also reported by *B* and vice versa. For example, for maven projects, 54% of Snyk's reported vulnerable dependencies were also reported by WhiteSource. Conversely, 68% of WhiteSource's reported maven dependencies were also reported by Snyk. Figure 2a, 2b demonstrates non-overlap in dependencies through a Venn diagram for three representative tools. We can not show such a heat map for all unique vulnerabilities, as we were unable to cross-reference non-CVEs across tools. However, we also found non-overlap over reported CVEs as well across tools. Figure 2c, 2d demonstrates non-overlap in CVEs for OWASP DC, Snyk, and WhiteSource.

Tools detected vulnerable dependencies across all scopes and depths: Table 4 shows a breakdown of scan results per dependency scope and what portion of the reported vulnerable dependencies are introduced directly by OpenMRS. We find that reported vulnerabilities are mostly introduced through transitive dependencies, except for Dependabot and WhiteSource. The latter two tools

assist GitHub projects in automatically fixing the vulnerable dependencies (by upgrading to a safer version), which may explain the high rate of direct dependencies in their reporting.

> We find SCA tools to vary widely in the reporting of known vulnerabilities, for both Maven and npm dependencies.

5.2 Why do the tools differ on vulnerability reporting?

We list the reasons we characterized (with no particular order) through manual analysis behind the differences in the tools' results:

(1) **OWASP DC and WhiteSource detected JavaScript dependencies in Maven projects:** The Maven projects in OpenMRS can also contain front-end JavaScript files. OWASP DC was able to identify dependencies from JavaScript files such as *jquery, handlebars*. These dependencies are not resolved by Maven itself or declared in any dependency manifest file. Besides OWASP DC, only WhiteSource detected JavaScript dependencies in Maven projects. In total, 42 JavaScript dependencies were found by OWASP DC, while WhiteSource found 20.

(2) **Only OWASP DC reported vulnerabilities in internal dependencies:** As mentioned in Section 3.2, OpenMRS can have internal dependencies which were reported only by OWASP DC. OWASP DC reported 200 dependencies which are OpenMRS projects. However, these 200 dependencies contain only 14 CVEs and 6 non-CVEs. OpenMRS projects are divided into many sub-modules and OWASP DC reports the same vulnerability separately for each sub-module, which results in an inflation of reported dependencies.

(3) **Same vulnerabilities can be repeated over multiple packages:** We observe tools may report the same vulnerability across many related packages, such as dependent packages of a vulnerable 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 as NVD simply lists the whole *spring-frameowrk* as affected by this CVE. Conversely, OWASP DC detected functions of npm packages as individual dependencies. In the *lodash* package, OWASP DC detected 31 functions, such as *lodash._baseassign* and *lodash._reevaluate*, separately besides the package itself, and repeated the same 7 vulnerabilities for each of them while other tools simply reported the *lodash* package as vulnerable. Prior work reported that relying on the Common Platform Enumeration (CPE) identifier that comes with CVE data may be a reason behind inaccurate vulnerability to package mapping [32]. For example, OWASP DC reported the same 17 CVEs for *activeio-core*, *activemq-core*, and *kahadb* as they all map to the same CPE identifier while other tools only reported *activemq-core*.

(4) **Tools may have different mapping of vulnerability to affected versions of packages:** Incorrect mapping of a vulnerability to the affected version range of a package

may result in inaccurate alerts. For example, in *commons-beanutils:1.7.0*, OWASP DC, WhiteSource, and Commercial A reported CVE-2014-0114 and CVE-2019-10086 while MSV reported only CVE-2014-0114 ; Dependabot reported only CVE-2019-10086; and Snyk reported no CVEs at all. To investigate this difference, we looked into Snyk's vulnerability database [15] 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 and therefore, considers the version OpenMRS uses as free of these vulnerabilities. Similarly, Dependabot 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.
Similarly, in the npm ecosystem, CVE-2018-1000620 was detected by all tools except Snyk for *cryptiles:0.2.2*. We found that Snyk's database lists [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.

(5) **Dependabot reported transitive dependencies through *lock* files:** We notice that Dependabot typically only detect direct dependencies. In the two cases where Dependabot reported transitive dependencies were 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. Further, no other tools reported dependencies from *lock* files except Dependabot, which detected 15 vulnerable dependencies from lock files.

(6) **Commercial B only reported vulnerabilities in dependencies under use during runtime:** As Commercial B tracks dependencies through interaction testing as explained in Section 4.2, the tool only reported dependencies that were under use by OpenMRS during integration testing, which explains the low count of dependencies reported.

(7) **The state of CVEs may result in differences in tools' results:** After a CVE is published, the CVE may become *reserved*, *disputed*, or *rejected* based on new information [3]. We observed that the state of CVEs may be one possible reason behind differences in CVE reporting, as SCA tools need to be timely updated and verify the changes in CVE states. For example, CVE-2019-10768 and CVE-2020-7676 in npm projects 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. The other CVE is *awaiting re-analysis* (subject to further changes) which may be a possible reason they are not incorporated by the latter tools. Further, we found four rejected CVEs to be reported by WhiteSource and Snyk which were not reported by other tools.

(8) **Tools can 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 dependencies that were reported to have non-CVEs by multiple tools, and found that each tool reported non-CVEs that were not reported by any other tools in the study set.

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 (Vulnerable 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 5: Code analysis based prioritization metrics: Vulnerable code reachability analysis

For example, we observe the following cases in *angular:1.6.1*: 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 the other tools. npm audit did not report any of these non-CVEs. We noticed similar differences in non-CVEs for other packages as well, e.g. *lodash, ws*.

> We categorize 8 reasons behind differences in vulnerability reporting among the studied SCA tools, such as inconsistency in vulnerability to affected package version mapping.

5.3 RQ2: What metrics are presented by the SCA tools to aid in the risk assessment of dependency vulnerabilities?

When a vulnerability lies in a dependency, the risk of the vulnerability may need to be determined by how the application uses the dependency – that is – in the context of the dependency. We have observed that the studied SCA tools reported several metrics in scan reports to aid in such contextual assessment. We characterize these metrics into five categories:

5.3.1 Code analysis-based metrics. Tools may analyze source code or binaries to infer dependency usage and vulnerability reachability. Three of the tools, Steady, Commercial A, and B, use code analysis-based metrics for Java language:

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 reachable from the dependant application through static and/or dynamic analysis. Steady and Commercial A provides reachability analysis for each vulnerability in dependency.

Steady constructs static call graphs of an application to infer reachability, referred to as *potentially executable* (static analysis). Steady also looks at the executability traces through unit testing to determine if the vulnerable code is *actually executed* (dynamic analysis). Commercial A, similarly perform static analysis to identify vulnerable call chains – that is – the call chain from the application code that reaches the vulnerable method of the dependency.

Table 5 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 by the dependant application. Further, Steady found only 2.1% of the alerts were *potentially executable* and 1.6% of the alerts were *actually executed*. However, we found a 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 found the vulnerable code to be actually executed, static analysis did not find any part of the dependency containing the vulnerability to be in use at all. 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 dependency can actually be reached from application source code specifying 93 distinct call chains.

Static analysis, such as call graph construction for Java, is known to have limitations [46]. The effectiveness of dynamic analysis, such as Steady's is also dependant on having a good test-suite and test coverage. We see that OpenMRS projects reach only around 20% test coverage in Steady. The limited test coverage may have affected Steady's findings.

Dependency Usage: The client application may only use a subset of the functionalities offered by a dependency. The code proportion of a dependency used by an application may indicate the probability of a vulnerability being reachable. Steady and Commercial B reports how many classes out of the total available are used in a dependency. For example, Commercial B found 203 out of 414 classes (49%) for *spring-web* and 790 out of 4,414 (17.9%) classes for *groovy-all* to have been used by OpenMRS.

5.3.2 Package Based metrics: The characteristics of the dependency package itself may indicate the risk associated with it.

Package security rating: Commercial B provides a letter grade on their assessment of the security of a package. The tool calculates the security rating of a package based on its age, count of released versions, and number of known vulnerabilities. Out of the 17 packages being identified as vulnerable by Commercial B, 16 have an *F* rating while one has a *D* rating.

5.3.3 Dependency characteristics based metrics: The scope and depth of the dependency may indicate the risk of the vulnerability it contains in the context of the application.

Dependency scope: For Maven projects, only Steady reported the scope for each dependency. For npm projects, Snyk and npm audit mentions the dependency scope.

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

5.3.4 ***Vulnerability based metric:*** The characteristics of the vulnerability itself can be used in assessing risk. We found three types of information provided by the tools:

Severity: The industry standard for rating the severity of vulnerability is the Common Vulnerability Scoring System [2] (CVSS), which are publicly available for CVEs. For the non-CVEs, Snyk and Commercial A also present a CVSS score. However, Dependabot and npm audit present a 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*

Available exploits: The availability of known exploits may contribute in assessing the risk for a vulnerability in a dependency. For each vulnerability, Snyk provides information on whether an exploit is publicly available. For the 310 Snyk vulnerabilities, Snyk reports that 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. 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 count of search hits within the past 30 days for the CVE identifier.

5.3.5 ***Confidence in alert validity:*** SCA tools may provide a confidence rating for each alert, which may aid developers in prioritizing auditing.

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

> Our tool survey highlights the research need for a common risk measurement framework specifically for dependency vulnerabilities that can be adopted by the SCA tools.

6 DISCUSSION

Below we discuss observations and implications from our work:

Resolving dependencies only through dependency manifest files may not provide all the open source code in use. We find that OWASP DC and WhiteSource detect JavaScript dependencies in Maven projects through source code scanning which are not declared in the Maven dependency file. This observation sheds light on the importance of scanning source code, binaries, and deployment environments to resolve all the open source code used by an application . Further, projects can use code fragments from open source packages [29] that should also be identified in order to report known vulnerabilities. Future research is necessary

to understand how reliably we can identify all the open source components used in a deployed application.

Accurate mapping of vulnerability to affected versions of packages should be ensured by the tools to avoid false positives: We showed examples in Section 5.2 on how inconsistency in vulnerability to package mapping can result in differences between tools' results. We also find inconsistencies in what version range is listed as affected for a specific CVE by different tools. Our findings highlight the importance of maintaining the accuracy of the tools' vulnerability databases.

Non-CVEs should get reported to CVE database to establish their validity and cross-tool vulnerability mapping. We find that SCA tools report known vulnerabilities in dependencies that do not have a CVE identifier. To understand why the non-CVEs might not have been incorporated into the CVE database, we look at their publication date. Of the 53 non-CVEs reported by Snyk, 41 were published before 2020; while of the 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. Furthermore, we see 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.

Developers may employ more than one tool to leverage different vulnerability databases: The reporting of unique non-CVEs by the tools highlights the existence of known vulnerabilities in the wild, not necessarily tracked by a centralized database. SCA tools may have delay in incorporating such non-CVEs, if not completely missing them. Prior work has shown that unawareness of the vulnerability is a major reason developers do not update vulnerable dependencies [33]. Therefore, our findings suggest developers may use multiple SCA tools to be timely informed of all the known vulnerabilities.

Tools should suggest fix options while explaining the risk of any potential backward incompatibility. Out of the studied tools, Snyk, Dependabot, npm audit, and WhiteSource offers automated fix suggestions. However, prior work has found that fear of breaking change is one of the primary reasons developers do not want to update vulnerable dependencies [21]. Tools may provide a more in-depth analysis on what code change is there with a certain version change in a dependency and if there is any possibilities of introduction of regression bugs.

Developers may need to evaluate security risk of dependency vulnerabilities case-by-case basis: Prior research has shown that not all vulnerabilities in dependency may be relevant for the client application [38, 39]. We find two tools to offer reachability analysis for each dependency vulnerability. However, future research is required to evaluate if developers in the real world actually use such metrics and find them useful or not. The lack of a common framework for risk assessment among SCA tools suggest that developers will have to evaluate the vulnerability alerts case-by-case basis based on their expertise on the project codebase and how the dependencies are being used by the specific project.

7 LIMITATIONS

We evaluate 9 SCA tools on one web application at a certain release point, which poses a threat to the generalizability of our findings. However, OpenMRS consists of 44 projects with 547 Maven and 2,213 npm dependencies, making our case study suitable for a comparative evaluation. Further, the single case study enables us to look in-depth with manual analysis on why the tools' results differed. However, our findings may not generalize to other package ecosystems, such as Ruby or Rust. Similarly, any other application with a difference in dependency management may have yield different findings. For example, the application that actively maintains its dependencies updated will have a few vulnerable dependencies overall and therefore, would not show large differences among tools' results. However, that would not invalidate the differences we observed for the studied SCA tools on OpenMRS. Another threat to the external validity involves the selection of the SCA tools. We explain our decision criteria in Section 4.1. While we are unable to cover all existing SCA tools, we do not claim the findings we have in Section 5.2 and 5.3 to be exhaustive.

Another limitation of our study involves the absence of ground truth. In the context of SCA alerts, there can be three steps for building such a ground truth - a) determining all the open source dependencies in use; b) determining the correctness of the reported vulnerability data; c) determining exploitability of the dependency vulnerabilities. These steps are nontrivial to perform manually for a real software and may be more suitable with a synthetic test subject [28]. Regarding exploitability, no tools except Commercial B claims to filter out dependencies on any use criteria, while Steady and Commercial A only offers additional analysis to aid in such contextual assessment. Therefore, exploitability would not be a fair comparison criteria for the studied SCA tools, and out-of-scope for this study. Similarly, how developers respond to alerts from SCA tools is also not in the scope of this study. Further, for RQ1, we do not conduct any statistical test to measure significance in difference, as we only perform a single case study.

8 RELATED WORK

The dependency network of package ecosystems and the presence of known vulnerabilities in dependencies have been studied in the literature [27, 31]. Decan et al. [26] 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 dependent packages do not get fixed even when the fix is available. [30] and [35] also found around one-third of the packages in the npm network to have at least one vulnerable dependency, while [30] found that *context use of the module* and *breaking changes* are potential reasons for not resolving these dependencies. The potential impact of the vulnerabilities in dependency has also been studied. Zapata et al. [49] 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 dependent applications did not actually use the vulnerable code. The study also finds that the dependent applications that do not use the vulnerable code take longer to migrate to new versions of a dependency. To detect dependencies where the vulnerable code is actually used by the including application, Ponta et

al. [40–42] proposed a *code-centric* and *usage-based* approach based on which the tool Steady is developed. Paschenko et al. [38] discuss the over inflation problem when reporting dependencies with unexploitable vulnerabilities. In another work, Paschenko et al. [39] interviewed developers on dependency management. The study found that developers think SCA tools generate many *irrelevant* and *low-priority* alerts, and may even rely on social channels than SCA tools for vulnerability reporting. Developers recommended SCA tools to *report only relevant alerts, work offline, and be easily integrated into company workflow*. The literature focuses on the importance of contextual assessment of vulnerable dependencies. To the best of our knowledge, there has been no study yet evaluating and comparing the existing SCA tools. The closest to our work is a recent study by Ponta et al. [42] where the authors compared their research tool Steady with OWASP Dependency-Check. The study compares the two tools over a sample of alerts generated on Java applications. The comparison was performed from the perspective of the reachability of a vulnerability in the dependency. The study finds both tools to have their unique findings. The study also finds Steady to have no false positives but a few false negatives while OWASP DC has non-negligible false positives. However, the study focuses on evaluating the detection capabilities of Steady based on its reachability analysis, whereas our study aims to provide a comprehensive comparison of existing SCA tools for both Java and JavaScript dependencies.

9 CONCLUSION AND FUTURE WORK

We evaluate 9 SCA tools on a large web application composed of Maven (Java) and npm (JavaScript) projects. We find that the tools vary across a wide range in the count of reported unique vulnerabilities and the dependencies that contain these vulnerabilities. Evidences in our findings suggest that accuracy, up-to-dateness, and completeness of vulnerability database is the key strength of an SCA tool. While building automation technologies for continuous monitoring of vulnerability data from open source package ecosystems is a future research need, developers at the moment may leverage multiple SCA tools in order to not miss any known vulnerability. Further, SCA tools should be able to pick up open source components beyond what is declared in the dependency manifest files, if any. We find that tools can provide code-analysis based metrics to assess the risk of dependency vulnerabilities. However, the effectiveness of such analysis needs to be evaluated and how developers in the real world responds to vulnerable dependencies should be studied. We have also seen tools to provide non code-analysis based metrics, such as package security rating, vulnerability exploits, and popularity. Prior work has found developers to rely on social channels to assess the risk of newly found vulnerabilities in open source packages [39]. As developers may get overwhelmed with frequent SCA alerts and patching may require extensive regression testing [44], future study is required on what metrics and information we can provide developers with to aid them in assessing and prioritizing the fix of vulnerable dependencies.

10 ACKNOWLEDGEMENTS

REFERENCES

[1] About semantic versioning. https://docs.npmjs.com/about-semantic-versioning.
[2] Common vulnerability scoring system. https://en.wikipedia.org/wiki/Common_Vulnerability_Scoring_System.
[3] Cve states. https://cve.mitre.org/cve/identifiers/.
[4] Eclipse steady 3.1.14 (incubator project). https://eclipse.github.io/steady/about/.
[5] Github advisory database. https://github.com/advisories.
[6] How does dependency-check work? https://jeremylong.github.io/DependencyCheck/general/internals.html.
[7] Mitre cve datagbase. https://cve.mitre.org/.
[8] National vulnerability database. https://nvd.nist.gov/vuln.
[9] Npm security advisories. https://www.npmjs.com/advisories.
[10] npm security advisories. https://www.npmjs.com/advisories.
[11] Openmrs around the world. http://guide.openmrs.org/en/.
[12] Openmrs reference application distribution. https://wiki.openmrs.org/display/docs/OpenMRS+Reference+Application+Distribution.
[13] Openmrs sdk. https://wiki.openmrs.org/display/docs/OpenMRS+SDK.
[14] Snyk open source security management. https://support.snyk.io/hc/en-us/articles/360000925438-What-does-Snyk-access-and-store-when-scanning-a-project-.
[15] Snyk vulnerability db. https://snyk.io/vuln.
[16] Sonatype oss index. https://ossindex.sonatype.org/.
[17] Steady vulnerability dataset. https://github.com/SAP/project-kb.
[18] Victims software vulnerability scanner. https://blog.victi.ms/.
[19] Whitesource bolt for github. https://github.com/apps/whitesource-bolt-for-github.
[20] Whitesource vulnerability database. https://www.whitesourcesoftware.com/vulnerability-database/.
[21] 0patch.com. Security patching is hard. https://0patch.com/files/SecurityPatchingIsHard_2017.pdf.
[22] 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.
[23] Benji Catabi-Kalman. Why do organizations trust snyk to win the open source security battle? https://snyk.io/blog/why-snyk-wins-open-source-security-battle/.
[24] Steven P Crain. Open source security assessment as a class project. *Journal of Computing Sciences in Colleges*, 32(6):41–53, 2017.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] Ibrahim Haddad. An open guide to evaluating software composition analysis tools, 2020.
[30] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? 2015.
[31] 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.
[32] Sean Kinzer. Using cpes for open-source vulnerabilities? think again. https://www.veracode.com/blog/managing-appsec/using-cpes-open-source-vulnerabilities-think-again.
[33] 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.
[34] 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.
[35] 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.
[36] 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].
[37] Top OWASP. Top 10-2017 the ten most critical web application security risks. *OWASP_Top_10-2017_% 28en*, 29, 2020.
[38] 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.
[39] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. *Proc. of CCS*, 20.
[40] 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.
[41] 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.
[42] 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.
[43] Apache Maven Project. "introduction to the dependency mechanism". http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.
[44] 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.
[45] Syed Zain Rizvi, Philip WL Fong, Jason Crampton, and James Sellwood. Relationship-based access control for openmrs. *arXiv preprint arXiv:1503.06154*, 2015.
[46] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. ICSE, 2020.
[47] Synopsys. 2021 open source security and risk analysis report. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html, 2021.
[48] Inger Anne Tøndel, Martin Gilje Jaatun, Daniela Soares Cruzes, and Laurie Williams. Collaborative security risk estimation in agile software development. *Information & Computer Security*, 2019.
[49] 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.
[50] 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.