Adversarial analysis of software composition analysis tools

Ekaterina Ivanova[1][0009−0006−4094−6943], Natalia Stakhanova[1][0000−0003−1923−319X], and Bahman Sistany[2]

[1] University of Saskatchewan, Saskatoon, Canada
frj039@usask.ca, natalia@cs.usask.ca
[2] NAV Canada, Ottawa, Canada
bahman.sistany@navcanada.ca

Abstract. With the widespread use of third-party code in software projects, Software Composition Analysis (SCA) tools emerged in order to help developers and security specialists automate the process of vulnerability detection within dependencies. Among SCA tools, the most common dependency detection techniques are metadata-based. However, there has not been a comprehensive evaluation of metadata-reliant SCA tools in regard to their resilience against metadata manipulations. To bridge this gap, we conducted a thorough evaluation of 5 state-of-the-art metadata-reliant SCA tools across 11 attack scenarios, each crafted to demonstrate a particular manifest feature, bundling, or dependency modification. Our findings reveal a concerning lack of resilience against metadata manipulations among these tools, with subtle modifications easily influencing their detection capabilities. Our findings not only uncover the limitations of existing metadata-based approaches but also offer valuable insights for SCA tool researchers, developers, and users.

Keywords: Software composition analysis · Dependency · Vulnerability

1 Introduction

In software development, third-party open-source code, in the form of *dependencies*, is commonly used to facilitate the development process. It has been estimated that dependencies typically constitute a significant proportion, approximately 80%, of the overall codebase in an average software project [22]. However, with great convenience comes great risk, as highlighted by OWASP identifying known vulnerabilities within dependencies as a top ten security concern in 2017 [20]. To address the inherent risks associated with third-party dependencies, Software Composition Analysis (SCA) emerged as a software security framework. SCA aims to identify and enumerate dependencies and their associated vulnerabilities within software projects. This task is increasingly being automated through a growing array of open-source and commercial SCA tools. However, the majority of these tools rely on metadata for dependency detection. Considerable attention has been devoted to examining the variances in tools' detection techniques and their performance [5, 12, 32]. Yet, the question of their robustness against attacks has received no attention. The recent vulnerability discovered in XZ Utils in Linux showed that there is a dire need to have reliable SCA tools. Hence, we need to understand the susceptibility of these tools to adversarial data manipulation.

In this work, we aim to address this gap. We present a series of attacks against software composition analysis of source code. Our attacks aim to prevent metadata-reliant SCA tools from correctly identifying included dependencies and their associated vulnerabilities. We achieve this by employing a variety of manifest feature, bundling, and dependency modifications.

We conduct a series of experiments to evaluate the efficacy of our attacks. To ensure a ground truth dataset, we constructed a comprehensive suite of 13 Maven projects, each using a distinct method of introducing dependencies. Each project contains 29 direct dependencies, all sourced from the top 100 most popular Java dependencies listed on the Maven Central Repository[3], with each dependency containing at least one associated vulnerability.

As our targets, we choose five well-known, widely used, and previously studied SCA tools that rely on metadata for their dependency detection. We conduct a thorough adversarial assessment of the performance exhibited by these tools, all of which showed little resilience against metadata manipulations.

We demonstrate that our attacks considerably affect all selected SCA tools. The attacks not only trigger *false negatives* but also cause what we will refer to as *false negative-positives*, where a tool mistakenly overlooks a legitimate dependency while flagging an incorrect one due to a particular metadata manipulation.

In summary, we made the following main contributions:

- Adversarial SCA. We present a series of attacks against software composition analysis of source code.
- Test suite. We built a test suite with a clearly defined ground truth for evaluating SCA tools' performance on Maven projects in the presence of metadata modifications and manipulations. The test suite is publicly available[4].
- Large-scale evaluation. We conducted a quantitative assessment of state-of-the-art SCA tools reliant on metadata for dependency detection in order to gauge their resilience against metadata manipulations. We manually analyzed differences in SCA tools' findings, investigating the underlying reasons and identifying vulnerabilities within their approaches, thereby enhancing our understanding of their limitations and potential areas for improvement.

The paper is organized as follows: Section 2 explains terminology and Section 3 discusses related research. Section 4 outlines our methodology. In Section 5, we provide an overview of the evaluation study, including information about the SCA tools we examined. Section 6 presents our findings. Finally, Section 7 concludes the paper, summarizing our key findings and discussing their implications.

[3] `https://mvnrepository.com/popular`
[4] `https://github.com/hinat0y/SCA-test-suite`

2 Background

In the context of Java software development, a **dependency** refers to open-source third-party code, in the form of a package, used by a client application. Each dependency is uniquely identified by a combination of a group ID, artifact ID, and version, such as *junit:junit:4.12*. While the definition of a dependency may be subject to debate [21], for the purposes of this study, we define a dependency as a specific version of a unique group ID and artifact ID pairing. This means that dependencies with identical versions within a single package are treated as separate entities. For example, *org.springframework:spring-context:5.3.12* and *org.springframework:spring-web:5.3.12* will be considered separate dependencies.

Maven is a package manager for Java projects. Maven manages a project's dependencies through *pom.xml* files, which we refer to as *manifest files*. The dependencies declared inside a project's manifest files are considered *direct dependencies*. However, these direct dependencies may, in turn, rely on additional packages necessary for their proper functioning. These additional dependencies, known as *transitive dependencies*, are automatically imported by Maven. By default, Maven only allows a single version of a dependency to be present inside a project. In cases where compatibility issues arise — such as when different versions of the same dependency are imported transitively — Maven approximates a solution using the "nearest definition" heuristic [9].

3 Related work

The detection of dependencies and corresponding vulnerabilities has been long studied [5, 6, 12, 26, 32]. Numerous studies looked at the spread and evolution of vulnerabilities in software packages [1,10], analysis of vulnerabilities in dependencies [2, 17]. Analysis of SCA tools so far has been primarily focused on accuracy and scalability of performance. Ponta et al. [26] conducted a comparative analysis of two SCA tools, Eclipse Steady and OWASP Dependency-Check, aiming to evaluate their performance in identifying vulnerabilities. Their study demonstrated the advantages of Eclipse Steady's code-centric approach over OWASP Dependency-Check's metadata-based approach, particularly in its capacity to reduce false positives. Imtiaz et al. [12] evaluated the performance of 9 SCA tools on a large web application, OpenMRS, that consisted of 43 Maven and 5 npm projects. Their study focused on comparing how these tools detected vulnerabilities and found differences in their approaches to dependency detection and vulnerability reporting. They concluded that the accuracy of the vulnerability database significantly influences the performance of SCA tools. Dann et al. [5] conducted an empirical study into the four common modifications through which dependencies can be introduced into a project: re-compilation, re-bundling, metadata removal, and re-packaging. Their evaluation, involving 6 SCA tools, aimed to quantify the impact of such modifications. None of the studied SCA tools could handle all four types of modifications. Dietrich et al. [6]

investigated the impact of shading and cloning on the accuracy of 4 SCA tools. Their research showed a noticeable negative impact of these dependency modifications on the performance of these tools. Subsequently, they introduced their own lightweight approach for detecting shaded dependencies. Zhao et al. [32] explored the compatibility of 6 SCA tools with various Maven dependency features and configurations, outlining their respective strengths and limitations in supporting diverse project settings.

Our study is complementary to previous research primarily due to its focus. While existing studies centred on evaluating the performance, accuracy and scalability of various dependency detection techniques, our work takes a different approach. We propose a series of attacks and assess metadata-reliant SCA tools' robustness against them.

4 Methodology

4.1 Threat model

For our series of attacks, we assume an adversary has a white-box access to the SCA tools, i.e., the adversary has an understanding of the features analyzed, the tools, and the employed analysis approach. Hence, an adversary can craft software package data to prevent the SCA tool from correctly identifying the included dependencies and vulnerabilities. The adversary's ultimate objective is to expand the attack surface by incorporating vulnerable, malicious, or illegal dependencies within software application while remaining undetected.

4.2 Adversarial SCA

SCA tools employ various techniques for dependency analysis, including static and dynamic analysis of source code [8, 9, 25, 26], binary analysis [7, 13], and snippet analysis [18]. However, by far the most common approach is metadata scanning. Unlike other approaches, metadata scanning does not analyze the dependency content directly but instead focuses on its metadata, e.g., manifest files, file names, digests, or timestamps [5,23]. This approach has numerous benefits, i.e., it can be easily automated, it is cost-effective and fast, as it does not rely on extensive databases. Prior research has highlighted its limitations, particularly when metadata is absent [5] or when an SCA tool provides an insufficient coverage of metadata combinations [32]. In our study, we explore this angle and adopt an attack strategy that targets metadata.

We design metadata manipulation scenarios that introduce subtle metadata alterations to provoke false positive and false negative detection by SCA tools. In the field of SCA, the terms false positive and false negative are a debated topic. Some researchers suggest that a dependency should be considered vulnerable only if its vulnerable code is reachable [9, 25, 31] or exploitable [4, 11, 16] from the client application. However, to align our findings with prevalent practices in state-of-the-art SCA tools, we will employ a broader definition. A **false positive** is a dependency that gets incorrectly flagged by an SCA tool, despite not being included in the project folder. Conversely, a **false negative** is when a vulnerable dependency is present but is not detected by the SCA tool.

In this work, we introduce the phenomena of **false negative-positive dependencies** where, due to specific manipulations, a tool wrongly identifies an absent dependency while overlooking a present vulnerable dependency. The primary distinction between a false positive or a false negative dependency and a false negative-positive dependency lies in the deliberate intent of an adversary. False positives usually arise from errors within dependency detection algorithms. On the other hand, false negative-positives can only occur when an adversary purposefully exploits a vulnerability inherent in a SCA tool. Moreover, simply causing false negatives might raise suspicions if the SCA tool detects no or few dependencies. This could prompt the team to switch tools or inspect the project manually. We assume that adversaries aim to create a false sense of security, avoiding suspicion while expanding the attack surface of a project. For false negative-positives, we only consider dependencies, as we assume the goal of an adversary is to create a false sense of security by minimizing the number or severity of reported vulnerabilities.

4.3 Attack strategies

We approach all attack strategies from an adversarial perspective. The designed attack strategies can be categorized into three groups: *manifest feature modifications*, *bundling modifications*, and *dependency modifications*. Manifest feature modifications are limited to alterations within a manifest file and rely on Maven for the installation and resolution of dependencies. Bundling modifications explore techniques where multiple dependencies are compressed into a single file, known as an Uber-JAR, and included inside a project. Lastly, dependency modifications cover the manual installation of dependencies. Instead of Maven automatically downloading a dependency, it adds a pre-compiled dependency to system folders. As our attacks are focused on manipulating metadata, the functionality of each project is not affected by these modifications.

Manifest feature modifications:
Attack Scenario 1: Version as variable. Dependency versions are defined as variables inside the *<properties/>* section of the manifest file, while group IDs and artifact IDs remain listed directly inside each *<dependency/>* tag.
Attack Scenario 2: Dependency management. Dependencies with their group IDs, artifact IDs, and versions are initially declared inside the *<dependencyManagement/>* section of the manifest file. However, the *<dependencyManagement/>* section is a declaration, not an invocation of dependencies. Therefore, within the subsequent *<dependencies/>* section, dependencies are then invoked without specifying their versions. Instead, the project relies on the version declarations provided in the *<dependencyManagement/>* section to resolve and include the dependencies.
Attack Scenario 3: Profiles. All dependencies are listed inside the default

profile inside the *<profiles/>* section of the manifest file. Within this profile, dependencies are included with their respective group IDs, artifact IDs, and versions. The *<profiles/>* section in a Maven manifest file allows for defining sets of configuration values, including modifying or adding dependencies, that can be activated or deactivated selectively during the project-building process.
Attack Scenario 4: Parent and child manifest (version as variable). In this attack scenario, the Maven project has a modular structure where dependencies and their versions are managed separately. Within the parent manifest file, dependency versions are declared inside the *<properties/>* section. Meanwhile, in the child manifest file representing an individual module, dependencies are invoked indirectly using variables defined in the parent manifest. This modular approach is normally utilized for maintainability and facilitates easier version control, as changes to dependency versions can be applied universally across all modules by updating the parent manifest.
Attack Scenario 5: Parent and child manifest (group ID as variable). Here, similar to Attack Scenario 4, the Maven project has a modular structure. However, rather than specifying versions of all dependencies inside the *<properties/>* section of the parent manifest, group IDs are defined instead. While this modification may raise suspicion as it is quite unconventional, it is permissible within the Maven framework. Maven permits the use of variables throughout the manifest, and an adversary might employ this technique to intentionally confuse an SCA tool.

Bundling modifications:
Attack Scenario 6: Uber-JAR. All direct and transitive dependencies are bundled into a single jar file, commonly known as an Uber-JAR, with a random name. This file is then manually installed into the project, including all necessary dependencies inside a single package. To achieve this, we used the Maven Assembly Plugin[5] to create an Uber-JAR with dependencies.
Attack Scenario 7: Shaded Uber-JAR. Shading is a technique commonly used by Java developers to avoid incompatibility issues between dependencies by repackaging them and modifying their package and class names. This process transforms original class names, such as *org.package.class*, into a shaded version — *org.shaded.package.class.* In this attack scenario, we shaded all direct and transitive dependencies inside the project. To accomplish this, we used the Maven Shade Plugin[6], which created a shaded Uber-JAR containing all dependencies. This shaded Uber-JAR was then manually installed into the project.
Attack Scenario 8: Bare Uber-JAR. In this attack scenario, we reused an Uber-JAR file with dependencies from a previous attack scenario (Attack Scenario 6). However, we stripped away all metadata typically relied upon by SCA tools. This included removing *pom.xml* files, *pom.properties* files, and the entirety of the *META-INF* folder. We also updatied the timestamps of all files within this binary. This modified Uber-JAR, with missing metadata and altered

[5] `https://maven.apache.org/plugins/maven-assembly-plugin/`
[6] `https://maven.apache.org/plugins/maven-shade-plugin/`

file timestamps, was then manually installed into the project.
Attack Scenario 9: Uber-JAR with modified metadata. In this attack scenario, we again reused an Uber-JAR with dependencies from Attack Scenario 6. However, rather than removing metadata entirely, we modified it. In each existing *pom.xml* and *pom.properties* file, we strategically changed the version of a vulnerable dependency. Specifically, we replaced the vulnerable version with a non-vulnerable, or if such a version was not available, with the latest version. Rather than overwriting entire *pom.xml* files by downloading them from the Maven Central Repository, we focused only on modifying the version information within the existing files. This modification aimed to simulate a scenario where an adversary manipulates metadata to conceal vulnerabilities within dependencies.

Dependency modifications:
Attack Scenario 10: Manually installed with a modified group ID. Dependencies are most often manually installed into the project because of patching. Patching a dependency traditionally involves modifying its source code before integrating it into a project. However, to investigate the constraints of metadata-reliant approaches, we refrained from modifying any dependency source code. Instead, we downloaded the jar files directly from the Maven Central Repository. Using the *mvn install:install-file* command, we installed these dependencies to the project. However, we introduced a deliberate typo in the group ID of each dependency. For instance, instead of using the correct group ID *org.springframework.boot*, we intentionally misspelled it as *org.springframeworkboot*. The rationale behind this modification was to simulate a scenario where an adversary intentionally introduces subtle errors in the metadata of manually included dependencies. Such deliberate misspellings could potentially cause SCA tools to overlook the dependency, all while evading detection. It is important to note that due to the manual installation, the installed dependencies did not automatically include their transitive dependencies.
Attack Scenario 11: Manually installed with a non-vulnerable version. Similar to the previous attack scenario, we manually installed dependency jars into the project. However, instead of making slight modifications to a dependency's group ID, we replace the dependency's version with either a non-vulnerable version or, if such a version is unavailable, with the latest version. We try to simulate a scenario where an adversary intentionally includes a vulnerable dependency while evading detection through metadata manipulation — specifically, by changing dependencies' versions.

4.4 Evaluation datasets

Developing a comprehensive test suite is crucial to effectively assess the adversarial resiliency of SCA. However, the availability of such test suites for SCA benchmarking proved to be scarce. Existing resources like DaCapo [3], Juliet

TestSuite [30], and OWASP Benchmark [19] primarily focus on application security testing and static analysis, lacking specific test cases for identifying vulnerabilities in dependencies. While benchmarks such as SourceClear [28], Achilles [5], and datasets developed by Zhao et al. [32] offered some relevant test cases, they fell short in providing detailed ground truth data or comprehensive coverage. For instance, while the SourceClear benchmark offered test cases for various programming languages and package managers, including Java and its package managers like Maven, Gradle, and Ant, its ground truth lacked specificity regarding particular dependencies and CVEs, making it unfit for assessing SCA tool adversarial resiliency. Similarly, the Zhao et al.'s datasets [32] lacked vulnerability data in its ground truth. On the other hand, while Achilles provided a well-defined ground truth, its coverage was very limited (7 dependencies and 16 CVEs) as its focus was on some dependency modifications, completely excluding manifest features. Furthermore, existing test suites assumed benign intentions from software creators, neglecting possible adversarial attempts to mislead SCA tools. Thus, recognizing the limitations of available resources, we constructed our own dataset covering 11 adversarial scenarios to assess the robustness of SCA tools against attacks.

For our dataset, we selected the Maven project manager for two primary reasons. Firstly, Maven projects are a common target in SCA tool benchmarking [5, 6, 12, 26, 32], offering a reliable basis for evaluation. Secondly, Maven's prominence as a preferred package manager among Java developers [14] is coupled with its inherent complexity and extensive feature set, making it an ideal environment for exploring metadata manipulation scenarios.

To construct a reliable dataset for our analysis, it was critical to select suitable projects and dependencies with known vulnerabilities. To ensure the relevance of our dataset, we selected from the 100 most widely used dependencies available on the Maven Central Repository[7]. We cross-referenced each chosen dependency with the National Vulnerability Database (NVD)[8] and OSSIndex[9] to identify any known vulnerabilities. The NVD is the primary repository of publicly accessible vulnerability management data. Utilizing the Common Vulnerabilities and Exposures (CVE) system, each vulnerability disclosed publicly receives a unique CVE identifier for tracking purposes within the NVD. However, the NVD is known to have shortcomings, including inaccurate and missing data [24, 27]. Sonatype OSSIndex is an alternative vulnerability database that we used to supplement NVD records.

Databases may also list vulnerabilities without an assigned CVE identifier. In this work, only vulnerabilities with assigned CVE identifiers are included in the analysis, and dependencies with assigned CVEs are referred to as *vulnerable dependencies*.

Out of the top 100 Java libraries, only 29 were found to have a CVE linked to at least one version of the dependency. To get a list of all transitive depen-

[7] `https://mvnrepository.com/popular`
[8] `https://nvd.nist.gov/`
[9] `https://ossindex.sonatype.org/`

dencies being introduced by the initial set, we created a basic Maven project and executed the *mvn dependency:tree* command. We also verified whether any known vulnerabilities were associated with these dependencies. Table 7 lists all selected dependencies alongside their respective versions and associated CVEs. Each entry in the table covers all unique CVEs linked to a specific version of a dependency according to the NVD and OSSIndex, including those associated with its transitive dependencies, which are written in gray. Table 8 lists all selected (direct) dependencies along with their transitive dependencies for scenarios where transitive dependencies are imported.

To create our test suite, we took the list of 29 selected dependencies and applied all 11 attack strategies outlined earlier to this list. Each strategy was implemented within a separate Maven project, with its own manifest file. Each project includes all 29 selected dependencies, along with any transitive dependencies. Additionally, to establish a consistent baseline for comparison across all attack scenarios, we added two baseline scenarios: one for manifest feature and bundling modifications and another – for dependency modifications. We then hosted these 13 Maven projects on GitHub.

4.5 Tools

In the concluding phase of our analysis, we proceed with an empirical evaluation of our series of attacks and investigate the robustness of metadata-reliant SCA tools in a series of experiments. For our assessment, we chose the following SCA tools: OWASP Dependency-Check[10], Dependabot[11], Grype[12], OSV-Scanner[13], and Snyk[14]. Since each tool presents its findings in a different format, after the evaluation, we conducted a manual examination of all tools' results.

OWASP Dependency-Check. This tool utilizes what it refers to as Analyzers to extract details from the files. The collected data, referred to as Evidence, includes vendor, product, and version of each dependency, including transitive dependencies. For example, the JarAnalyzer extracts information from manifest files and file names inside jar files. Importantly, Dependency-Check focuses on runtime metadata, typically found in the META-INF folder and its subfolders [6]. We used its Maven plugin to scan all our projects.

Dependabot. Dependabot scans manifest files to identify dependencies and their corresponding versions, although it does not include transitive dependencies and does not scan binaries. We hosted all 13 of our Maven projects on the first author's GitHub and activated Dependabot alerts. However, since Dependabot alerts report vulnerability data only, we used GitHub Dependency Graph[15],

[10] `https://owasp.org/www-project-dependency-check/`
[11] `https://github.com/dependabot`
[12] `https://github.com/anchore/grype`
[13] `https://github.com/google/osv-scanner`
[14] `https://snyk.io/`
[15] `https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph`

Table 1: Baseline analysis results

	Baseline 1										Baseline 2									
	Present		**Detected**		**TP**		**FP**		**FN**		**Present**		**Detected**		**TP**		**FP**		**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl
Dependency-Check	**68**	**87**	68	87	68	87	0	0	0	0	**29**	**67**	29	59	29	59	0	0	0	8
Dependabot	**68**	**87**	29	45	29	42	0	3	39	45	**29**	**67**	29	45	29	42	0	3	0	25
Grype	**68**	**87**	29	45	29	42	0	3	39	45	**29**	**67**	29	45	29	42	0	3	0	25
OSV-Scanner	**68**	**87**	28	45	28	42	0	3	40	45	**29**	**67**	1	2	1	2	0	0	28	65
Snyk	**68**	**87**	68	71	68	63	0	8	0	24	**29**	**67**	29	47	29	42	0	5	0	25

dp - number of dependencies, vl - number of vulnerabilities, TP - true positive, FP - false positive; FN - false negative

an SBOM generating tool offered by GitHub, to double-check dependency detection. This approach helped us identify cases where Dependabot could not detect dependency versions or when it misidentified non-vulnerable versions.

OSV-Scanner. The OSV-Scanner tool examines manifest files, SBOM files, and git directories, but it presently lacks support for Maven's transitive dependencies or binary files. We used their command-line tool and conducted scans across all directories containing our Maven projects.

Grype. Grype is as an SCA tool primarily used for scanning container images [15, 29]. However, it also supports scanning directories containing Maven projects and Java binary files. It does not resolve transitive dependencies by default. We installed its command-line tool and conducted scans across all directories containing our Maven projects and Uber-JARs with dependencies. Since Grype reports vulnerability data only, we used Syft[16], an SBOM generating tool often used alongside Grype, to double-check dependency detection. This approach helped differentiate cases where a tool failed to detect a dependency from cases where it incorrectly identified a dependency, such as reporting a non-vulnerable version.

Snyk. Snyk is a commercial SCA tool that analyzes manifest files for dependency and vulnerability information. It resolves transitive dependencies with its knowledge base, although it lacks support for scanning binary files. We used its free trial version and integrated it with our GitHub repositories.

5 Evaluation results

Baseline analysis. Our baseline analysis aims to provide the foundation for evaluating the impact of subsequent attacks, and to give us a clear reference point for comparison. Two baseline scenarios present ideal scenarios with all dependencies explicitly listed inside the <*dependencies*/> section, with their correct group ID, artifact ID, and version specified directly inside each <*dependency*/> tag. In baseline 1, Maven's automatic resolution mechanism ensures that transitive dependencies are included in the project as required. On the other hand, baseline 2 includes only direct dependencies and their unique associated CVEs due

[16] `https://github.com/anchore/syft`

to the fact that manually installed dependencies do not import transitive dependencies by default. To signify that a dependency is manually installed, a suffix such as "_patch" is added to the end of each dependency's version. Baseline 1 is designed as a reference point for manifest feature modification and bundling modification attacks, and baseline 2 serves as a reference point for dependency modification attacks.

The results of baseline analysis are presented in Table 1. One of the tools, OWASP Dependency-Check, showed perfect accuracy, both in terms of detection of dependencies and the corresponding vulnerabilities in both baselines. Snyk showed the second best results for Baseline 1 detecting all dependencies accurately and identifying 72% (63 out of 87) vulnerabilities. Dependabot, Grype, and OSV-Scanner showed a similar mediocre performance for baseline 1, i.e., they correctly identify above 42% of the dependencies (29 out of 68 for Dependabot, Grype, and 28 out of 68 for OSV-Scanner) and 48% of vulnerabilities (42 out of 87).

The presence of false negatives in the results from Dependabot, Grype, and OSV-Scanner can largely be attributed to their limitations in detecting transitive dependencies. Specifically, 20 out of 45 (44%) of these false negatives arose because these tools failed to identify transitive dependencies and the vulnerabilities associated with them. Unlike these tools, Snyk relies on a knowledge database to estimate transitive dependencies, which helps in addressing this gap to some extent. On the other hand, OWASP Dependency-Check integrates directly with the project, enabling it to inspect system folders for both direct and transitive dependencies.

Furthermore, the considerable number of false negatives reported by these tools highlights another issue: variations in the vulnerability databases they rely on. Specifically, Dependabot, Grype, and OSV-Scanner each identified 25 false negatives due to these discrepancies, while Snyk reported 24. These discrepancies arise because the tools use different vulnerability databases. As a result, even if all dependencies are correctly identified, the resulting vulnerability reports can differ significantly.

Interestingly, in baseline 2, despite using the same vulnerability database as our ground truth, Dependency-Check reported 8 false negative vulnerabilities even though it accurately detected all dependencies. A manual investigation revealed that these false negatives were due to the limitations of the tool's algorithm, not discrepancies in the vulnerability data. Similarly, Snyk, Dependabot, and Grype performed notably well for baseline 2, successfully identifying all dependencies and detecting 63% (42 out of 67) of the vulnerabilities. On the other hand, OSV-Scanner's performance was much worse, as it recognized only 1 out of 29 dependencies and identified just 2 out of 67 vulnerabilities in baseline 2.

Results of manifest feature modifications. As results in Table 2 show, in both attack scenario 1 and 2 tools performed comparable to the baseline. Both Dependency-Check and Snyk were able to correctly detect all dependencies, while Dependabot, Grype, and OSV-Scanner produced a large number of false negatives (57-58%).

Table 2: SCA results for manifest feature modifications attacks

	Present		Scenario 1								Scenario 2								Scenario 3							
			Det.		**TP**		**FP**		**FN**		**Det.**		**TP**		**FP**		**FN**		**Det.**		**TP**		**FP**		**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl
Dependency-Check	**68**	**87**	68	87	68	87	0	0	0	0	68	87	68	87	0	0	0	0	68	87	68	87	0	0	0	0
Dependabot	**68**	**87**	29	45	29	42	0	3	39	45	29	45	29	42	0	3	39	45	29	45	29	42	0	3	39	45
Grype	**68**	**87**	29	45	29	42	0	3	39	45	25*	77	25*	31	0	46	33	56	0	0	0	0	0	0	68	87
OSV-Scanner	**68**	**87**	28	45	28	42	0	3	40	45	28	45	28	42	0	3	40	45	0	0	0	0	0	0	68	87
Snyk	**68**	**87**	68	71	68	63	0	8	0	24	68	71	68	63	0	8	0	24	68	71	68	63	0	8	0	24

** - dependency versions were not detected, dp - number of dependencies, vl - number of vlnerabilities*

Table 3: SCA results for manifest feature modifications attacks

	Present		Scenario 4								Scenario 5							
			Detected		**TP**		**FP**		**FN**		**Detected**		**TP**		**FP**		**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl
Dependency-Check	**68**	**87**	68	87	68	87	0	0	0	0	68	87	68	87	0	0	0	0
Dependabot	**68**	**87**	29*	0	29*	0	0	0	39	87	0	0	0	0	0	0	68	87
Grype	**68**	**87**	25*	77	25*	31	0	46	33	56	0	0	0	0	0	0	68	87
OSV-Scanner	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	68	87
Snyk	**68**	**87**	68	71	68	63	0	8	0	24	68	71	68	63	0	8	0	24

** - dependency versions were not detected, dp - number of dependencies,*
vl - number of vulnerabilities, TP - true positive, FP - false positive, FN - false negative

While Grype reported on most of direct dependencies, it could not identify their versions, which led to a compromised analysis in terms of reported vulnerabilities. This implies that metadata-dependent tools make some effort to cover at least some of the package manager's features.

In attack scenario 3, the inability of OSV-Scanner and Grype to detect any dependencies or vulnerabilities demonstrates a significant weakness in the coverage provided by these tools with regard to manifest features. This highlights the potential for adversaries to exploit blind spots if all possible manifest features are not adequately addressed.

Attack scenarios 4 and 5 experiment with multi-module structure of a Maven project. As shown in Table 3, some of the tools struggled with this configuration. In attack scenario 4, OSV-Scanner failed to detect any dependencies. Moreover, although Dependabot and Grype could detect dependencies when the child manifest referenced versions from the parent manifest, their analysis remained limited. For example, in this scenario, Dependabot failed to report any associated vulnerabilities, while Grype identified vulnerabilities linked to unidentified versions, similar to its performance in attack scenario 2. Taking advantage of this weakness, adversaries may reference group IDs instead of versions in a child manifest, rendering three tools — Dependabot, Grype, and OSV-Scanner — ineffective in identifying dependencies with a simple modification, which is demonstrated in attack scenario 5. Similar to other scenarios, OWASP Dependency-Check was able to successfully report all dependencies and vulnerabilities in both scenarios. Snyk performed similarly well, showing a slightly worse (72% detection accuracy) performance on vulnerabilities (63 out of 87).

Results of bundling modifications. Among the analyzed tools, only two were able to handle Uber-JARs: OWASP Dependency-Check and Grype. To assess

Table 4: SCA results for bundling modifications attacks

			Scenario 6								Scenario 7							
	Present		**Detected**		**TP**		**FP**		**FN**		**Detected**		**TP**		**FP**		**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl
Dependency-Check	**68**	**87**	16	41	16	41	0	0	52	46	16	41	16	41	0	0	52	46
Dependabot	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	68	87
Grype	**68**	**87**	16	33	16	32	0	1	52	55	16	33	16	32	0	1	52	55
OSV-Scanner	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	68	87
Snyk	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	68	87

dp - number of dependencies, vl - number of vulnerabilities, TP - true positive;
FP - false positive, FN - false negative

Table 5: SCA results for bundling modifications attacks

			Scenario 8								Scenario 9								
	Present		**Detected**		**TP**		**FP**		**FN**		**Detected**		**TP**		**FP**		**FNP**	**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	dp	vl
Dependency-Check	**68**	**87**	0	0	0	0	0	0	68	87	16	0	0	0	16	0	16	68	87
Dependabot	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	0	68	87
Grype	**68**	**87**	0	0	0	0	0	0	68	87	16	0	0	0	16	0	16	68	87
OSV-Scanner	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	0	68	87
Snyk	**68**	**87**	0	0	0	0	0	0	68	87	0	0	0	0	0	0	0	68	87

dp - number of dependencies, vl - number of vulnerabilities, TP - true positive;
FP - false positive, FP - false negative-positive, FN - false negative

their performance with Uber-JARs, we created four attack scenarios. As shown in Table 4, in attack scenario 6, both OWASP Dependency-Check and Grype managed to identify only 16 dependencies out of a total of 68 (24%) and 41 (47%) and 33 (38%) vulnerabilities respectively. Upon closer examination, we discovered that only these 16 dependencies imported *pom.xml* files with clearly defined group IDs, artifact IDs, and versions inside. This observation led us to speculate that both tools conduct binary analysis by examining manifest files and extracting information from them, rendering other metadata irrelevant. Attack scenario 7 further affirmed this observation, as we shaded all direct and transitive dependencies, yet the modified dependency paths within the binary files did not influence the performance of either tool. Both Dependency-Check and Grype identified the same dependencies and vulnerabilities as in attack scenario 6.

In attack scenario 8, we removed all metadata within the *META-INF* folder, including *pom.xml* and *pom.properties* files, as well as updated all timestamps. None of the tools were able to detect any dependencies or vulnerabilities. On the other hand, in attack scenario 9, we deliberately modified the versions inside every *pom.xml* and *pom.properties* file present in the Uber-JAR generated for attack scenario 6. Both OWASP Dependency-Check and Grype not only failed to identify any present dependencies or vulnerabilities, they ended up misidentifying all 16 dependencies (0% detection accuracy), reporting modified versions instead of actual versions. In other words, both tools' findings were entirely false negative-positives.

Results of dependency modifications. In attack scenarios 10 and 11 (Table 6), we experimented with manual installation of dependencies, a scenario that fre-

Table 6: SCA results for dependency modifications attacks

			Scenario 10								Scenario 11							
	Present		**Detected**		**TP**		**FP**		**FN**		**Detected**		**TP**		**FP**		**FNP**	**FN**	
	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	vl	dp	dp	vl
Dependency-Check	**29**	**67**	29	68	29	53	0	15	0	14	29	26	0	26	29	0	29	29	41
Dependabot	**29**	**67**	0	0	0	0	0	0	29	67	29	3	0	3	29	0	29	29	64
Grype	**29**	**67**	29	45	29	42	0	3	0	25	29	45	29	42	0	3	0	0	25
OSV-Scanner	**29**	**67**	0	0	0	0	0	0	29	67	0	0	0	0	0	0	0	29	67
Snyk	**29**	**67**	0	0	0	0	0	0	29	67	29	2	0	1	29	1	29	29	66

dp - number of dependencies, vl - number of vulnerabilities, TP - true positive;
FP - false positive, FP - false negative-positive, FN - false negative

quently occurs when developers need to update or customize a dependency. Typically, developers download the source code of the dependency, implement modifications, and compile it. Developers with benign intentions normally maintain accurate metadata, opting for small modifications such as appending a word to the end of the dependency's version (e.g., *1.2.3* to *1.2.3_patch*) inside a manifest file [5]. However, our objective was to examine the effects of maliciously altering this metadata to mislead the SCA tools.

In attack scenario 10, we intentionally misspelled the group IDs of each dependency within the project's manifest (e.g., *com.googleguava* instead of *com.google.guava*), and manually installed all dependencies accordingly. OSV-Scanner, Snyk, and Dependabot failed to detect any of the dependencies or vulnerabilities. On the other hand, OWASP Dependency-Check managed to successfully identify all dependencies and 53 of 67 vulnerabilities (79%). However, it is important to note that the modification did have an impact on the performance of the tool: it reported 15 false positive vulnerabilities. Grype showed a slightly worse performance by correctly identifying all present dependencies but only 42 vulnerabilities (63%).

In attack scenario 11, we modified the versions of each dependency to either non-vulnerable versions or, in cases where such versions were unavailable, to the latest ones. OSV-Scanner failed to detect any dependencies and vulnerabilities, while Grype correctly identified all dependencies and most vulnerabilities (63%). However, OWASP Dependency-Check, Snyk, and Dependabot only reported false negative-positives, detecting non-existent versions of dependencies based on the manipulated metadata taken from the project's metadata. While Snyk and Dependabot's reliance on analyzing the manifest file makes them susceptible to manifest manipulations, OWASP Dependency-Check's behaviour is particularly interesting. Its algorithm focuses on gathering "Evidence" from various sources, including mentions of dependencies found in metadata such as manifest files and file names. Although it appeared to correctly identify versions from system files, it prioritized the information inside the project's manifest, leading to false negative-positives. Interestingly, this only applied to vulnerability data sourced from the NVD, as data from OSSIndex accurately reflected the actual versions of dependencies, which led to at least partially correct vulnerability data (39% accuracy).

6 Discussion

Our experiments reveal the limitations of the existing metadata-reliant approaches employed by the state-of-the-art SCA tools.

The attack resilience of the existing metadata-based approaches is inadequate. In our evaluation, OWASP Dependency-Check stood out as the most flexible and accurate among the examined tools. It performed comparatively well when faced with various types of modifications. On the contrary, OSV-Scanner was the most susceptible to adversarial manipulations. Moreover, it struggled to analyze projects with both bundled and manually installed dependencies.

For attacks employing manifest feature modifications, OWASP Dependency-Check and Snyk showcased the most resilience by providing thorough coverage of diverse manifest configurations. In the context of bundling modification attacks, although their results were largely compromised, OWASP Dependency-Check and Grype were the only two tools that were able to analyze Uber-JAR files. Lastly, in the case of attacks via dependency modifications, Grype outperformed other tools by accurately identifying all included dependencies in both attack scenarios.

Our findings show the limitations of metadata-based approaches and demonstrate their overwhelming susceptibility to attacks. This indicates a need for alternative techniques for constructing SCA methods designed with robustness in mind. These limitations can potentially be mitigated through:

- *Choosing less manipulable metadata.* Metadata-based approaches offer obvious advantages. However, file names and manifest files can be easily modified in order to influence a tool's findings. It could be beneficial for metadata-reliant SCA tools to move away from metadata that can easily be modified or erased towards metadata that is more permanent and harder to manipulate such as file digests, checksums, file sizes and so on.
- *Ensuring metadata integrity.* Our findings demonstrate the fact that the state-of-the-art metadata-reliant SCA tools struggle with conflicting or incomplete metadata such as unidentified dependency versions. Implementing measures to handle such scenarios is crucial. Moreover, in the case of Uber-JAR analysis, SCA tools should perform integrity checks (such as checksums) for manifests to mitigate blatant manipulations.
- *Supplementing metadata analysis.* While metadata-based approaches outperform other approaches in some cases, SCA tools are likely to benefit from a multifaceted approach to dependency analysis. For example, there is strong reason to believe that static and dynamic call analysis can provide valuable insights in cases where metadata is either absent or unreliable [5].

Metadata-reliant SCA tools do not provide comprehensive coverage of dependency specifications. The inadequate detection of vulnerabilities is a direct result of inadequate recognition of dependencies. In particular,

– *Transitive dependencies.* In our experiment, three of the chosen tools failed to report on transitive dependencies. Although in real-life projects, the likelihood of transitive dependencies introducing vulnerabilities to the client application is relatively low, it does open the door for adversaries to strategically obscure these vulnerabilities. This scenario could result in the client application unknowingly utilizing vulnerable functions from a dependency, evading detection by SCA tools. We would argue it is imperative for metadata-based approaches to prioritize the inclusion of transitive dependencies in their analysis.
– *Manually installed dependencies.* Our findings show that manual installation of dependencies is another convenient avenue for adversaries to include vulnerable dependencies without being detected. Metadata-reliant SCA tools should make an active effort to detect dependencies regardless of the way they are being introduced to a project.
– *Uber-JARs.* Our experiment showed that support for Uber-JARs among metadata-reliant SCA tools is limited. While the practice of aggregating multiple dependencies into a single file is a common and usually benign practice, it also serves as a potential loophole for manipulating the findings of SCA tools, either to evade detection entirely or to prompt the reporting of wrong dependencies. Recognizing this challenge, it is imperative for tool developers to allocate additional resources towards enhancing the capabilities of their tools to effectively handle Uber-JARs.

SCA approaches do not appear to be tested for resilience against adversarial manipulations. At its core, SCA is a cybersecurity practice. Therefore, researchers must adopt a holistic perspective when evaluating approaches, focusing not only on their effectiveness, accuracy, and scalability but also on their resilience against malicious manipulations. In this way, researchers can enhance the trustworthiness of proposed solutions among users. Moreover, such an approach expands the range of scenarios in which they can be applied.

7 Conclusion

This paper presents a set of metadata manipulation attacks and a comparative analysis examining the robustness of 5 state-of-the-art metadata-reliant SCA tools. We developed a test suite that implemented 11 different attack scenarios, exploring a variety of feature manifest, bundling, and dependency modifications. Our findings highlight a concerning lack of resilience against metadata manipulations among all studied SCA tools, with their findings significantly influenced by minor modifications. We hope that our developed test suite and the insights described in this study will urge both researchers and developers to critically assess the robustness of SCA tools against adversarial approaches.

References

1. Alfadel, M., Costa, D.E., Shihab, E., Adams, B.: On the discoverability of npm vulnerabilities in node.js projects. ACM Trans. Softw. Eng. Methodol. **32**(4) (2023)
2. Alqahtani, S.S., Eghan, E.E., Rilling, J.: Tracing known security vulnerabilities in software repositories – a semantic web enabled modeling approach. Science of Computer Programming **121**, 153–175 (2016)
3. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. SIGPLAN Not. **41**(10), 169–190 (Oct 2006)
4. Chen, Z., Hu, X., Xia, X., Gao, Y., Xu, T., Lo, D., Yang, X.: Exploiting library vulnerability via migration based automating test generation. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Lisbon, Portugal (Apr 2024)
5. Dann, A., Plate, H., Hermann, B., Ponta, S.E., Bodden, E.: Identifying challenges for oss vulnerability scanners - a study test suite. IEEE Transactions on Software Engineering **48**(9), 3613–3625 (2022)
6. Dietrich, J., Rasheed, S., Jordan, A.: On the security blind spots of software composition analysis (2023)
7. Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W.: Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 2169–2185. CCS '17, Dallas, Texas, USA (Oct 2017)
8. Foo, D., Chua, H., Yeo, J., Ang, M.Y., Sharma, A.: Efficient static checking of library updates. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 791–796. ESEC/FSE 2018, Lake Buena Vista, FL, USA (Oct 2018)
9. Foo, D., Yeo, J., Xiao, H., Sharma, A.: The dynamics of software composition analysis (2019)
10. Germán Márquez, A., Varela-Vaca, A.J., Gómez López, M.T., Galindo, J.A., Benavides, D.: Vulnerability impact analysis in software project dependencies based on satisfiability modulo theories (smt). Comput. Secur. **139**(C) (2024)
11. Iannone, E., Nucci, D.D., Sabetta, A., De Lucia, A.: Toward automated exploit generation for known vulnerabilities in open-source libraries. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). pp. 396–400. Virtual, Spain (May 2021). https://doi.org/10.1109/ICPC52881.2021.00046
12. Imtiaz, N., Thorn, S., Williams, L.: A comparative study of vulnerability reporting by software composition analysis tools. In: Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). ESEM '21, Virtual, Italy (Oct 2021)
13. Jiang, L., An, J., Huang, H., Tang, Q., Nie, S., Wu, S., Zhang, Y.: Binaryai: Binary software composition analysis via intelligent binary source code matching. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24, Lisbon, Portugal (Apr 2024)
14. JRebel: 2021 java developer productivity report. https://www.jrebel.com/resources/java-developer-productivity-report-2021 (2021)

15. Kalaiselvi, R., Ravisankar, S., M, V., Ravindran, D.: Enhancing the container image scanning tool - grype. In: 2023 2nd International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA). pp. 1–6. Coimbatore, India (2023)
16. Kang, H.J., Nguyen, T.G., Le, B., Păsăreanu, C.S., Lo, D.: Test mimicry to assess the exploitability of library vulnerabilities. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 276–288. ISSTA 2022, Virtual, South Korea (Jul 2022)
17. Li, Q., Song, J., Tan, D., Wang, H., Liu, J.: Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 161–173. Virtual, Taiwan (Jun 2021)
18. Lightner, N.: Introducing synopsys ai code analysis api. https://www.synopsys.com/blogs/software-security/synopsys-ai-code-analysis-service.html#3 (2023)
19. Mburano, B., Si, W.: Evaluation of web vulnerability scanners based on owasp benchmark. In: 2018 26th International Conference on Systems Engineering (ICSEng). pp. 1–6. Sydney, NSW, Australia (Dec 2018)
20. OWASP: OWASP Top 10 Application Security Risks - 2017. https://owasp.org/www-project-top-ten/2017/Top_10 (2017)
21. Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F.: Vulnerable open source dependencies: Counting those that matter. In: Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM) (2018)
22. Pashchenko, I., Vu, D.L., Massacci, F.: A qualitative study of dependency management and its security implications. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 1513–1531. CCS '20, Virtual, USA (Nov 2020)
23. Pereira, D., Molloy, C., Acharya, S., Ding, S.H.H.: Automating sbom generation with zero-shot semantic similarity (2024)
24. Plate, H., Ponta, S.E., Sabetta, A.: Impact assessment for vulnerabilities in open-source software libraries. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 411–420. Bremen, Germany (Sep 2015)
25. Ponta, S., Plate, H., Sabetta, A.: 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). pp. 449–460. Madrid, Spain (Sep 2018)
26. Ponta, S.E., Plate, H., Sabetta, A.: Detection, assessment and mitigation of vulnerabilities in open source dependencies. Empirical Softw. Engg. **25**(5), 3175–3215 (Sep 2020)
27. Sabetta, A., Bezzi, M.: A practical approach to the automatic classification of security-relevant commits. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 579–582. Los Alamitos, CA, USA (Sep 2018)
28. SourceClear: Evaluation framework for dependency analysis. https://github.com/srcclr/efda (2020)
29. Tran, N.K., Pallewatta, S., Babar, M.A.: Toward a reference architecture for software supply chain metadata management (2023)
30. Wagner, A., Sametinger, J.: Using the juliet test suite to compare static security scanners. In: Proceedings of the 11th International Joint Conference on E-Business

and Telecommunications - Volume 4. p. 244–252. ICETE 2014, Vienna, Austria (Aug 2014)

31. Wu, Y., Yu, Z., Wen, M., Li, Q., Zou, D., Jin, H.: Understanding the threats of upstream vulnerabilities to downstream projects in the Maven ecosystem. In: Proceedings of the 45th International Conference on Software Engineering. p. 1046–1058. ICSE '23, Melbourne, Victoria, Australia (2023)
32. Zhao, L., Chen, S., Xu, Z., Liu, C., Zhang, L., Wu, J., Sun, J., Liu, Y.: Software composition analysis for vulnerability detection: An empirical study on java projects. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23). pp. 960–972. San Francisco, CA, USA (Dec 2023)

8 Appendix

Table 7: Dependencies and unique vulnerabilities (transitive in gray)

Dependency	**CVEs**
clojure-1.11.1	CVE-2024-22871
commons-beanutils-1.8.3	CVE-2014-0114, CVE-2019-10086
commons-collections-3.2	CVE-2015-6420
commons-io-2.4	CVE-2021-29425
fastjson-1.2.22	CVE-2022-25845, CVE-2017-18349
gson-2.8.6	CVE-2022-25647
guava-24.1.1-jre	CVE-2020-8908, CVE-2023-2976
h2-2.0.204	CVE-2022-45868, CVE-2018-14335, CVE-2022-23221, CVE-2021-42392
httpclient-4.5.11	CVE-2020-13956
jackson-databind-2.13.1	CVE-2022-42003, CVE-2020-36518, CVE-2022-42004, CVE-2023-35116
json-20200518	CVE-2023-5072, CVE-2022-45688
junit-4.12	CVE-2020-15250
kotlin-stdlib-0.6.22	CVE-2019-10103, CVE-2019-10101, CVE-2020-29582, CVE-2022-24329, CVE-2019-10102
log4j-core-2.17.0	CVE-2021-44832
logback-classic-1.1.11	CVE-2021-42550, CVE-2017-5929
logback-core-1.3.5	CVE-2023-6378
mysql-connector-java-8.0.11	CVE-2021-2471, CVE-2022-21363, CVE-2019-2692, CVE-2020-2875, CVE-2023-22102, CVE-2018-3258, CVE-2020-2934
okhttp-3.1.0	CVE-2018-20200, CVE-2023-0833, CVE-2016-2402, CVE-2021-0341, CVE-2023-3635
protobuf-java-3.21.5	CVE-2022-3509, CVE-2022-3171, CVE-2022-3510
retrofit-2.3.0	CVE-2018-1000850
scala-library-2.13.0	CVE-2022-36944
spring-beans-5.3.4	CVE-2021-22118, CVE-2021-22096
spring-boot-autoconfigure-3.0.3	CVE-2023-34055
spring-boot-starter-web-2.6.0	CVE-2023-20873, CVE-2023-20883, CVE-2023-22602, CVE-2023-41080, CVE-2022-42252, CVE-2022-23181, CVE-2022-41854, CVE-2023-45648, CVE-2022-38752, CVE-2022-34305, CVE-2022-29885, CVE-2023-42795, CVE-2022-38750, CVE-2023-28708, CVE-2023-46589, CVE-2022-45143, CVE-2022-1471, CVE-2022-38749, CVE-2021-43980, CVE-2023-44487, CVE-2022-25857, CVE-2022-38751
spring-context-5.3.12	CVE-2022-22950, CVE-2021-22060, CVE-2022-22968, CVE-2022-22965, CVE-2022-22971, CVE-2022-22970
spring-core-6.0.3	CVE-2023-20860, CVE-2024-22233, CVE-2023-20863, CVE-2023-20861
spring-web-6.1.3	CVE-2024-22262, CVE-2024-22243
spring-webmvc-6.0.10	CVE-2023-34053
testng-7.4.0	CVE-2022-4065

Table 8: Direct and transitive dependencies

Direct dependency	**Transitive dependency**
clojure-1.11.1	spec.alpha-0.3.218, org.specs.alpha-0.2.62
commons-beanutils-1.8.3	
commons-collections-3.2	
commons-io-2.4	
fastjson-1.2.22	
gson-2.8.6	
guava-24.1.1-jre	jsr305-1.3.9, error_prone_annotations-2.1.3, j2objc-annotations-1.1, checker-compat-qual-2.0.0, animal-sniffer-annotations-1.14
h2-2.0.204	
httpclient-4.5.11	commons-codec-1.11, commons-logging-1.2, httpcore-4.4.13
jackson-databind-2.13.1	jackson-annotations-2.13.1, jackson-core-2.13.1
json-20200518	
junit-4.12	hamcrest-core-1.3
kotlin-stdlib-0.6.22	kotlin-runtime-0.6.22
log4j-core-2.17.0	log4j-api-2.17.0
logback-classic-1.1.11	slf4j-api-1.7.22
logback-core-1.3.5	
mysql-connector-java-8.0.11	
okhttp-3.1.0	okio-1.6.0
protobuf-java-3.21.5	
retrofit-2.3.0	
scala-library-2.13.0	
spring-beans-5.3.4	
spring-boot-autoconfigure-3.0.3	spring-boot-3.0.3
spring-boot-starter-web-2.6.0	spring-boot-starter-2.6.0, spring-boot-starter-json-2.6.0, spring-boot-starter-tomcat-2.6.0, spring-boot-starter-logging-2.6.0, snakeyaml-1.29, jakarta.annotation-api-1.3.5, log4j-to-slf4j-2.14.1, jackson-datatype-jdk8-2.13.0, jackson-datatype-jsr310-2.13.0, jackson-module-parameter-names-2.13.0, tomcat-embed-core-9.0.55, tomcat-embed-el-9.0.55, tomcat-embed-websocket-9.0.55, jul-to-slf4j-1.7.32
spring-context-5.3.12	spring-aop-5.3.12, spring-expression-5.3.12
spring-core-6.0.3	spring-jcl-6.0.3
spring-web-6.1.3	micrometer-observation-1.12.2, micrometer-commons-1.12.2
spring-webmvc-6.0.10	
testng-7.4.0	jcommander-1.78, jquery-3.5.1