# Supply Chain Benchmark
# Leading Tool Comparison

**Luca Carettoni & Anthony Trummer**

DOYENSEC

# CONTENT

11

# About This Research Work

# INTRODUCTION

**Doyensec performed a side-by-side comparison of three popular Software Composition Analysis (SCA) tools (Semgrep, Snyk, and Dependabot) in order to evaluate their abilities to properly determine whether an application's third-party libraries, with known vulnerabilities, actually introduce an exploitable condition in the application. This included confirming that vulnerable library versions were not just included, but also that the vulnerable functions or configurations were actually used as described in the public disclosures. A high degree of accuracy in such an analysis is required to reduce the overall number of false positives, hence reducing the overall triaging effort required by professionals. Through manual analysis, we measured true and false positives and determined the level of effort required by a security team to investigate the tool findings.**

## Keywords

software supply chain security, software composition analysis (SCA), Dependabot, Semgrep, Snyk

## SUPPLY CHAIN SECURITY MANAGEMENT

Modern applications are commonly composed of a mix of custom code from its creators and a large number of open-source libraries, typically used to provide common functionality accelerating development timelines. While the creators of the application are responsible for its security, historically they have focused on the custom code, relying on the open-source community to discover vulnerabilities in the third-party libraries it uses.

Unfortunately, even when open-source libraries have identified and patched vulnerabilities, application maintainers lacked an efficient means to easily collect and track vulnerability information for the many libraries used in their applications. This situation led to tools which create databases of the vulnerabilities in many of the commonly used open-source libraries and have the ability to scan application source code to identify their use.

This evolution was an improvement over the previous state, but while the tools generated a lot of findings, they could not determine whether an exploitable condition actually existed. Due to the time required to prove or disprove the existence of each reported vulnerability, some organizations choose not to adopt this technology, or at least they may perceive a diminished value from it. A good analysis of the issues in Software Supply Chain Security can be found here[1].

The next step in the evolution of these tools looks to address this issue by mirroring some of the capabilities previously only available in Static Application Security Testing (SAST) tools. Specifically, they're now examining the details of how the dependencies are integrated into applications, rather than just the fact that the libraries are present. To do this, SCA tools have improved their abilities to understand how the code is constructed and how data flows through it. These modern tools now use parsers to read the code and determine whether the code is constructed in a vulnerable manner that matches the details in the public vulnerability disclosures. Ultimately, this serves to reduce the number of false positives reported to the end users.

There are a lot of pieces necessary to perform this type of in-depth analysis of application source code. An ideal solution for these tools would include:

- The ability to accurately detect the name and versions of libraries used
- A robust database of libraries with known vulnerabilities
- A robust database of rules for determining whether vulnerable libraries are used in a manner that introduces exploitable vulnerabilities
- A support infrastructure that supplies continuous updates to all its vulnerability databases
- The ability to run versioned scans to control which rules are run. This helps with automation, so builds don't suddenly start failing due to new rules (or potentially bugs in rules) and to know if new detections are from new rules or new code
- The ability to view all the database contents to know which vulnerabilities could be detected and what detection gaps exists
- Clear reporting on whether a vulnerable library is included in an exploitable manner or just matched by name and version (to alert teams to the potential for future issues)

[1] https://tldrsec.com/p/supply-chain-security-overview

## OBJECTIVES

Our research into SCA tools was specifically centered on their ability to **reduce false positives rates** through the addition of the more thorough code examination capabilities. We wanted to determine how the tools performed on real-world code and compare their results.

Ultimately, an improvement in false positive rates in these tools has the potential to improve their rate of adoption. Firstly, lowering false positive rates means those responsible for triaging the results should spend considerably less time validating them, once they are confident of their accuracy. This frees them to address other issues, potentially improving security in other areas as well. Secondly, when the tools gain trust, they theoretically should reduce the mean time to resolution for vulnerabilities, because the reported issues can be acted upon more quickly, rather than having to wait for manual validation. Furthermore, improving the false positive rates should lead to wider adoption, since higher rates are a major reason organizations don't implement these types of solutions. This in turn leads to less vulnerable applications being deployed. For all these reasons, we wanted to perform an analysis of not only the false positive rates, but the average time saved as a result.

Before diving into the analysis, an agreement is needed as to what is meant by false positives. In this examination, we would want to verify the following to confirm a vulnerability (true positive):

- The name of the package matches
- The version of the package matches
- The dependency is implemented:
  - Using a configuration consistent with the advisory and/or,
  - Making use of the vulnerable functionality described in the advisory

Consequently, any reported issues that didn't match **all** of these criteria would be considered a false positive.

To further limit the scope, the possible implementation environment, expected use cases, and/or the attackers' access to the vulnerable code paths were not factored into the evaluation. Additionally, no attempts were made to locate vulnerabilities outside of those collectively reported by the tools. It is therefore theoretically possible that an unreported vulnerability could be present that was missed by all the tools.

Conversely, it is also necessary to establish the definition of a true or false negative. When using something like the OWASP Benchmark Project, the answers are clear, because it is working with contrived code snippets whose vulnerabilities (or lack thereof) are already known. However, when testing against real-world code, there is no prior knowledge of its vulnerabilities. While it's reasonably possible to validate any true or false positives, assessing all the true or false negatives simply isn't practical.

## OUR RESEARCH METHODOLOGY

In order to benchmark the tools using real-world scenarios, we decided to use public OSS repositories written in JavaScript, Python, Ruby, or Java. We started by selecting 1000 of GitHub's "top-rated"[2] candidate repositories for each of the languages we wanted to test. Additionally, all candidate repositories had to have at least 5 commits in the last 10 months. From there, we cloned the repositories from the most recent commit that was at least 10 months prior. This was meant to give all tools the opportunity to have detection rules for known vulnerabilities in such codebases.

From our candidate pool, we applied the following selection criteria, based on the results from Semgrep's Supply Chain reachability analysis feature (later "Semgrep" when referencing the tool used):

- Must have vulnerabilities in "direct" dependencies (not in dependencies of the dependencies)
- Must have Critical or High severity vulnerabilities (as reported by Semgrep)
- Must have advisories from 2022 or 2023

This was meant to represent recent vulnerabilities that we would expect all organizations to consider as a potential security risk.

From these results, we selected 50 vulnerabilities that Semgrep claimed were "reachable" and 50 vulnerabilities that were claimed to be included, but were not "reachable" (i.e., the name and version matched, but no vulnerable code path existed). We recognize that a compromise was made due to the fact that there could have been false negatives that would have been counted against Semgrep, which would not be included in the candidate pool. Our acceptance of this tradeoff was mostly due to the fact that we were primarily focused on the true or false positives, rather than the negatives. Once the 100 vulnerabilities were selected, we scanned the same repos with Dependabot and Snyk as well.

Benchmarking was performed using  Semgrep Supply Chain, Snyk, and Dependabot, using their default configuration For transparency, we list some of the details regarding each of them.

- Dependabot
  - Dependabot's documentation[3] : *we have details of vulnerable functions for 79 Python advisories from the pip ecosystem*. This suggests that it only tests whether *"your code is calling vulnerable code paths"* for those issues, implying that the other findings (i.e., those in other languages) are solely from name and version matching.
  - Testing was performed by cloning the public repositories, enabling vulnerability alerts and retrieving the alerts via GitHub's APIs

- Semgrep Supply Chain
  - We only considered its "reachability" rules, since that was the focus and users could easily filter the results on this basis. This specifically excludes: parity, manual-review, and always reachable rules.
  - We used a Team tier license with the Pro Engine installed.
  - Version 1.37.0 of the CLI client was used for scanning.

[2] This was determined based on the number of stars a repository had received
[3] https://github.blog/2022-04-14-dependabot-alerts-now-surface-if-code-is-calling-vulnerability/

- The following CLI command was used:
  semgrep ci -supply-chain --json --suppress-errors --output *semgrep-out*

- Snyk (with reachable vulnerabilities scanning option enabled)[4]
  - Documentation states: *If a "no path found" status is given, do not assume that the vulnerability is totally unreachable.*[1] We have no way of distinguishing the reason for a "no path found" result and feel ultimately it is irrelevant to the end user, since their only recourse is manual validation either way.
  - Documentation states: *Reachable vulnerabilities analysis is available for **Java** (Maven and Gradle) with supported versions using the **GitHub** integration.*[4] This implies that the other findings (i.e., those in other languages) are solely from name and version matching. We used Snyk Open Source - 5 Team Contributing Developers Pack Monthly
  - Snyk requires all dependencies to be installed. This was done manually, then we executed Snyk with the appropriate package manager definition
  - Example command used:
    snyk test -package-manager=npm --json-file-output=snyk-out.json

All scans were performed between August 27th and 28th, with the tools updated at the time of testing.

The second phase of this project consisted of manually verifying the reported vulnerabilities from all three tools. Broadly, we started by examining the code and, through manual code review, validating whether a reported vulnerability was indeed a true positive or a false positive.

[4] https://docs.snyk.io/manage-risk/find-and-manage-priority-issues/reachable-vulnerabilities

## RESULTS

As mentioned above, we selected 50 vulnerabilities that Semgrep claimed were “reachable” and 50 vulnerabilities that were claimed to be included, but were not “reachable”.

Ultimately, this selection included 45 unique CVEs, across 11 GitHub repositories:

| | | |
|---|---|---|
| CVE-2022-0235 | CVE-2022-46175 | CVE-2023-25669 |
| CVE-2022-0686 | CVE-2023-2251 | CVE-2023-25670 |
| CVE-2022-21227 | CVE-2023-22578 | CVE-2023-25672 |
| CVE-2022-23476 | CVE-2023-22579 | CVE-2023-25673 |
| CVE-2022-24785 | CVE-2023-22794 | CVE-2023-25674 |
| CVE-2022-25878 | CVE-2023-24807 | CVE-2023-25675 |
| CVE-2022-25927 | CVE-2023-25658 | CVE-2023-25676 |
| CVE-2022-32224 | CVE-2023-25659 | CVE-2023-25801 |
| CVE-2022-3517 | CVE-2023-25660 | CVE-2023-25813 |
| CVE-2022-35937 | CVE-2023-25662 | CVE-2023-30629 |
| CVE-2022-35939 | CVE-2023-25663 | CVE-2023-30837 |
| CVE-2022-39299 | CVE-2023-25664 | CVE-2023-31146 |
| CVE-2022-41894 | CVE-2023-25665 | CVE-2023-32058 |
| CVE-2022-41900 | CVE-2023-25666 | CVE-2023-32059 |
| CVE-2022-41902 | CVE-2023-25668 | CVE-2023-36665 |

For transparency, we report the GitHub URLs and commit ids for the projects included in this evaluation at the time of testing:

- Java/Sentinel - https://github.com/alibaba/Sentinel.git
  dd1ba2725e06e2b9fbdb7d6aec41a421a01afc3e
- Java/openapi-generator - https://github.com/OpenAPITools/openapi-generator.git
  1984a31004e9db1b30928778da7bee33ab99fb3a
- JavaScript/artillery - https://github.com/artilleryio/artillery.git
  f1aced604e5c18483e3e60985ff30eb3d300f700
- JavaScript/codimd - https://github.com/hackmdio/codimd.git
  b55bf97024e03e052490e67b1b04b66324c55924
- JavaScript/next.js - https://github.com/vercel/next.js.git
  bba571a59c73ca26a9e7dba5dfbf683e1501f54f
- JavaScript/pencil - https://github.com/evolus/pencil.git
  47b238976c3e3ec1626b73cdae141be79c4c132c
- Python/brownie - https://github.com/eth-brownie/brownie.git
  86258c7bdf194c800ae44e853b7c55fab60a23ce
- Python/spleeter - https://github.com/deezer/spleeter.git
  bdc27726412dd296ae9d51434db402675565d129
- Python/streamlit - https://github.com/streamlit/streamlit.git
  87bb89a3b3ac77a5fd7d95c7dbc76335f01e23c1
- Ruby/gitlabhq - https://github.com/gitlabhq/gitlabhq.git
  51c18a25f2751911e134e73dbc946ee130fc6487

- Ruby/parallel - https://github.com/grosser/parallel.git
  2cee039fe145ae0a5058484d23ce9c141e46fb72

By scanning the very same projects at the same point in time with all tools under analysis, we identified 115 unique CVEs.

During our manual review process we tracked the time it took our engineers to manually validate the reported findings. Per our calculations, on average, it took **12.3 minutes** of focused time (free from other interruptions) per vulnerability, to determine its existence. This will obviously vary with the complexity of the code, the specific vulnerability and the skill and experience of the engineer performing the reviews.

Combining this information and the data below, we find that reviewing a project of this scale, an organization would expect to invest:

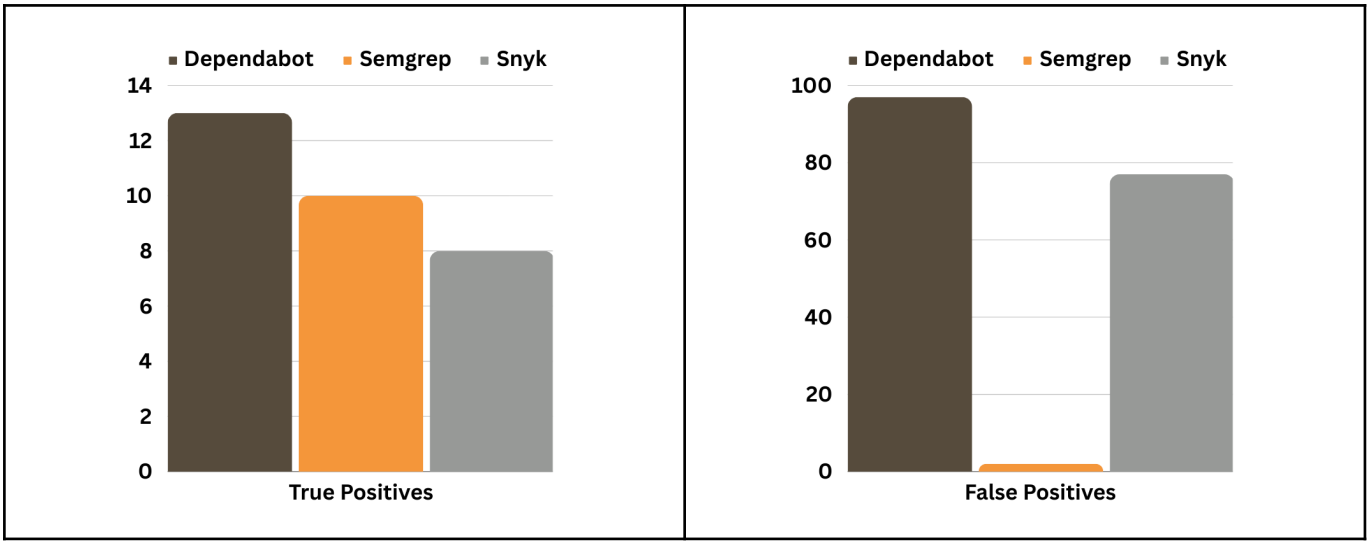
- Dependabot: 1353 minutes to validate the positive findings, with 12% being valid
- Semgrep: 148 minutes to validate the positive findings, with 83% being valid
- Snyk: 1046 minutes to validate the positive findings, with 9% being valid

It should be noted that these metrics will vary based on the specific circumstances and do not provide statistical significance.

The raw numeric results of our testing are as follows:

COMPARISON CHART

| TOOL | TRUE POSITIVES | FALSE POSITIVES |
|---|---|---|
| DEPENDABOT | 13 | 97 |
| SEMGREP | 10 | 2 |
| SNYK | 8 | 77 |




Summarizing the results of our analysis of these tools, we find the following:

- Dependabot discovered the largest number of true positives
- Dependabot reported the largest number of false positives
- Semgrep reported the least number of false positives

## OUR TAKEAWAYS

Different organizations will have different requirements, so there is no way to say one tool is best for everyone. An organization with a massive team and time to spare, with an extremely critical application (e.g., government, military) may have the desire to find every single vulnerability, regardless of the effort involved. A more resource-constrained organization with a higher risk tolerance might prefer only to receive true positives, even at the expense of missing some level of vulnerabilities.

For our “at-scale” analysis, Dependabot and Snyk were significantly more difficult to set up and use than Semgrep. Dependabot required custom instrumentation to leverage the integration with GitHub, while Snyk required installing all dependencies of the projects under analysis. That said, we feel in a real-world scenario, Dependabot is likely to be the easiest to use whenever projects are hosted on GitHub. Having said that, we don’t believe that such differences will be meaningful for long-term deployments.

Taking all of this into consideration, if we were to be selecting a tool today and we wanted a high signal-to-noise ratio, which could be trusted enough to deploy in a CI/CD pipeline, it seems Semgrep would be the clear choice.

Looking at the numbers, Dependabot and Snyk are automatically removed from consideration for automation in CI/CD on the basis of the high false positive counts alone. Even for manual analysis, the review of a large number of findings with a less than 15% accuracy rate is unlikely to justify the effort by an internal AppSec team.

For us, the additional findings from Dependabot aren’t worth the additional work, when compared to Semgrep, unless one is operating in an environment with excessive amounts of resources and high guarantees.

# ABOUT DOYENSEC

Doyensec was founded in 2017 by John and Luca who are its only stakeholders. The company exists to further the passion and focus of its creators. We aim to provide research-driven application security, enabling trust in our client's products and evolving the resilience of the digital ecosystem.

With offices in the US and Europe, Doyensec has access to a unique talent pool of security experts capable of providing worldwide consulting services.
We keep a small dedicated client base and expect to develop long term working relationships with the projects and people involved. We will find bugs, but we know that is just the first step in the process. At any stage of your security maturity, you can rely on Doyensec to solve your unique application security needs.

We value and rely on the following principles:

- **Passion**. We believe quality comes from passion and care. We love what we do, and continuously work on mastering our craft. Every engagement is finely executed with dedication and attention to details.
- **Expertise**. Our team has decades of experience in application security. We are industry leaders in penetration testing, reverse engineering, and source code review. Doyensec researchers have discovered numerous vulnerabilities in widely-deployed products, secured fortune 500 enterprises, advised startups and worked with tech companies to eradicate security flaws.
- **Focus**. Security craftsmanship is all about individual attention and delivering tailored security services and products. We concentrate on application security and do fewer things, better.
- **Research**. The fast changing landscape of technologies and security threats requires constant innovation. We are dedicated to providing research-driven application security and therefore invest 25% of our time in building security testing tools, discovering new attack techniques and developing countermeasures.

## ABOUT THIS RESEARCH WORK

This research is based upon work financially supported by Semgrep. Despite that, Doyensec had complete freedom on the research execution and publication. While Semgrep had the right to decide on whether to publish the effort in its entirety or just citing parts according to our Citation Guideline, under no circumstances has Doyensec adjusted the results represented in this publication.


DOYENSEC