Dependency Management Report

The best way to secure your software supply chain? It depends.

ENDOR LABS

Contents

Introduction

Software development continues to trend toward extensive use of third party components — known as dependencies — that free up a developer's time to work on business critical code. This includes:

Application dependencies
The libraries and frameworks that your first-party code relies on (including AI or ML models responsible for generating new first-party code).

Build-and-Deploy dependencies
All the things you rely on to turn your first-party code into a distributable application, such as GitHub Actions.

Operational dependencies
Components needed to operate your applications in production, such as container images.

Most dependencies are free and open source (OSS), saving the world an estimated $8.8 trillion in development costs. But OSS isn't without its own risks, and the use of generative AI as a development assistant is already spawning new challenges. Managing risks, including vulnerabilities, in OSS dependencies is a top concern for organizations seeking to secure the software development lifecycle (SDLC).

Our annual Dependency Management Report explores emerging OSS dependency trends to consider as part of an SDLC security strategy. In 2024, we answered several questions relating to application dependencies:

☑ How successful are AppSec teams at identifying dependencies and their vulnerabilities?
☑ Does public vulnerability data support successful researching and prioritizing of vulnerabilities?
☑ What gets in the way of remediating known vulnerabilities?
☑ How can software composition analysis (SCA) improve dependency management?

Executive Summary

First, what is a “dependency” anyway?

A ‘software dependency’ refers to external code or libraries that a software project requires to function properly. Most commonly, dependencies are recognized as third-party libraries, frameworks, or other software components that provide essential functionality without having to write it from scratch. At Endor Labs, we recognize that software development has changed enough that this definition needs to expand. That’s why we go beyond traditional dependencies to secure everything your code depends on, extending the definition of “dependency” to include tools we use to build, test, and operate applications.

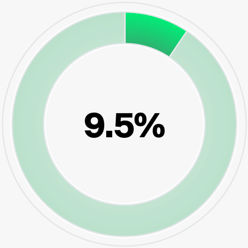
So, “dependency management” is what, then?

Dependency management involves ensuring that all required dependencies are correctly specified, resolved, and integrated into a project.

Traditional dependency management often focuses on identifying and cataloging 3rd party libraries based on manifest files, which list all required packages. This can lead to a superficial, or even inaccurate, understanding of how dependencies interact and impact the overall security of the software.

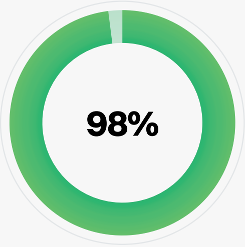
In contrast, Endor Labs employs a more comprehensive approach that includes examining which parts of the code actually utilize specific dependencies - we call this reachability analysis.

Dependency management all comes down to effective prioritization



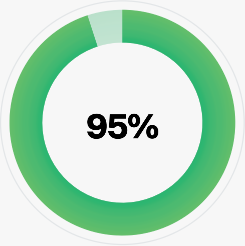
Less than 9.5% of vulnerabilities are exploitable at the function level

Function-level reachability is the most important prioritization capability.



Programs that combine reachability and EPSS see a noise reduction of 98%.

After reachability, EPSS is the second-most important prioritization technique.



95% of version upgrades contain at least one breaking change.

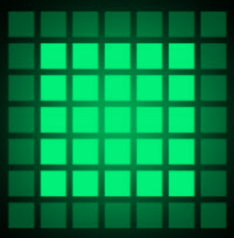
We found that remediation of vulnerabilities in dependencies requires a major version update 24% of the time. Minor updates can potentially break a client 94% of the time, and patches have a 75% chance of causing a break.

Prioritizing risks is only the first step; organizations need help prioritizing remediation efforts.

Dependency management is impossible with public advisory databases alone

Dependency risk management programs typically rely on open source or commercial tools to enable this analysis, but since many tools do not enrich public advisory data, defenders can not assume tools provide better quality data than a given advisory database.

25 days



is the median delay between public patch availability and advisory publication

Only 2%

of public advisories contain information about which library functions contain the vulnerability

1/4

of advisories have either incorrect or incomplete data that can lead to false positives and false negatives

Artificial Intelligence makes programming easier, but dependency management harder

1

Vulnerability reporting is less consistent for AI libraries to begin with

Reported vulnerability counts for a significant AI and ML library can vary by as much as 10% between public advisory databases.

2

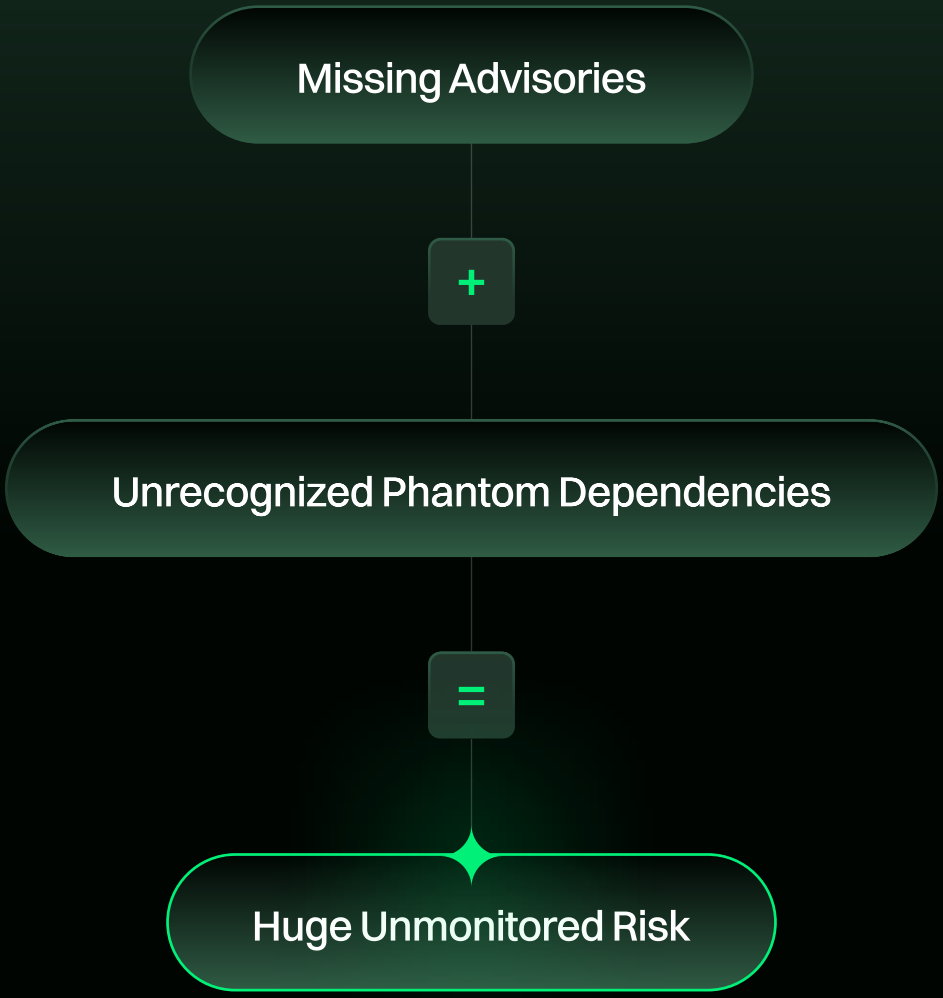
Phantom dependencies are more common in AI and ML software projects

AI software projects tend to be written in Python, a language notorious for including dependencies that are not declared in manifests.

3

56% of reported library vulnerabilities are in phantom dependencies for organizations with significant phantom dependency footprints

These factors cause a double-whammy for dependency management for AI-generated projects.



Missing Advisories

+

Unrecognized Phantom Dependencies

=

Huge Unmonitored Risk

1

Part 1: Identifying Dependencies & Their Vulnerabilities

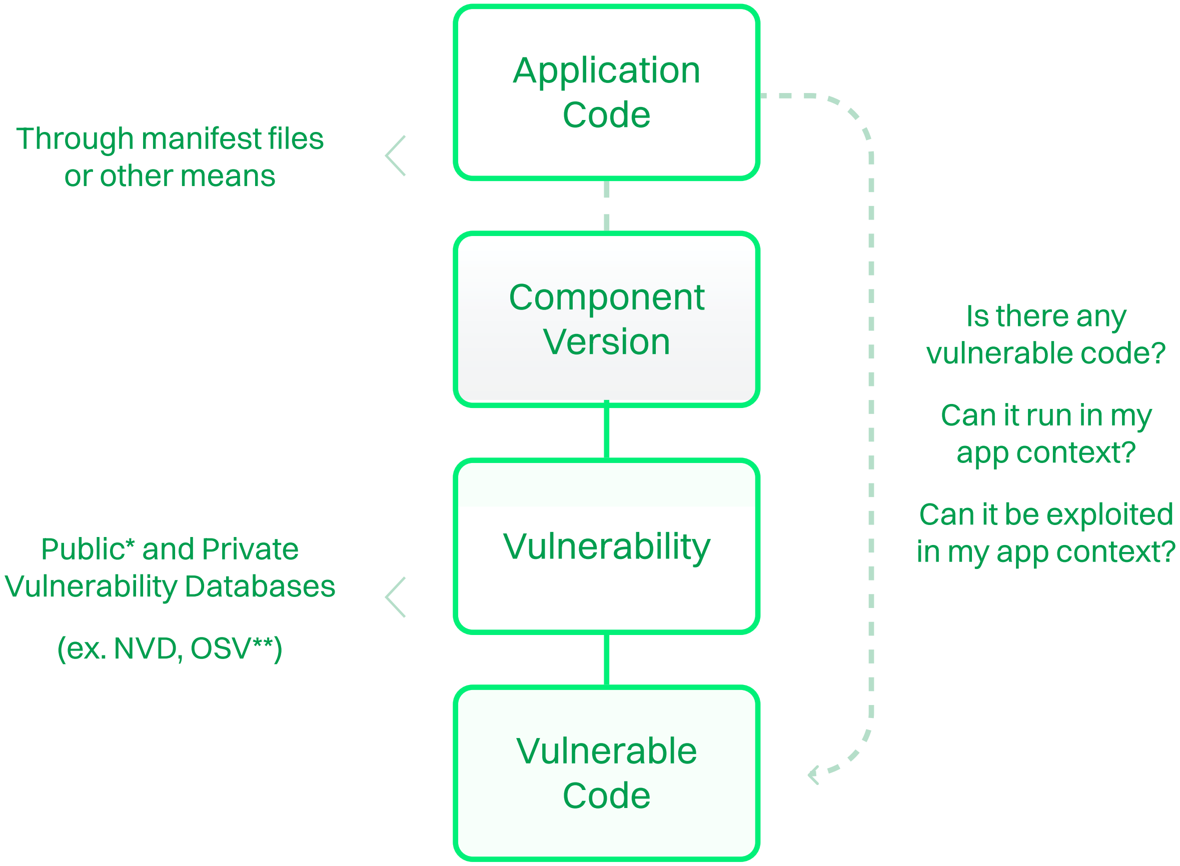
Finding Known-Vulnerable Code in Dependencies

TL;DR

- Technical challenges make it hard to link applications to vulnerable code in their dependencies.
- Dependency identification and the quality of vulnerability databases are key to avoiding false positives and false negatives.

Reachability analysis becomes an established technique to assess and prioritize vulnerabilities – be it in academic papers, open source tools or commercial solutions.

There are multiple types or definitions of reachability. At Endor Labs, what we mean by reachability analysis is to determine whether an application that includes a vulnerable function can be executed in a way such that the vulnerable function also gets executed – which is a prerequisite for a vulnerability to be exploitable.


Application Code
Through manifest files or other means
Component Version
Vulnerability
Public* and Private Vulnerability Databases
(ex. NVD, OSV**)
Vulnerable Code
Is there any vulnerable code?
Can it run in my app context?
Can it be exploited in my app context?

As illustrated by Figure 1, the question whether vulnerable open source code is included in some application is answered by linking (1) applications to the component versions it depends on, (2) components and their versions to vulnerabilities and (3) vulnerabilities to affected functions. If this link can be established, we can proceed using static program analysis to construct a call graph and determine the reachability of a known-vulnerable function.

Establishing this link, however, is hindered by a variety of problems – which will be one focus of this year's edition of the report:

- First, the challenge to establish all dependencies of an application, especially beyond what's managed through manifest files.
- Second, the difficulty to correctly identify affected components, component versions and functions through advisories. In other words, the correctness and timeliness of vulnerability databases, which has been a heatedly discussed topic in early 2024, when the NVD backlog suddenly increased due to "an increase in software and, therefore, vulnerabilities, as well as a change in interagency support" [NVD].

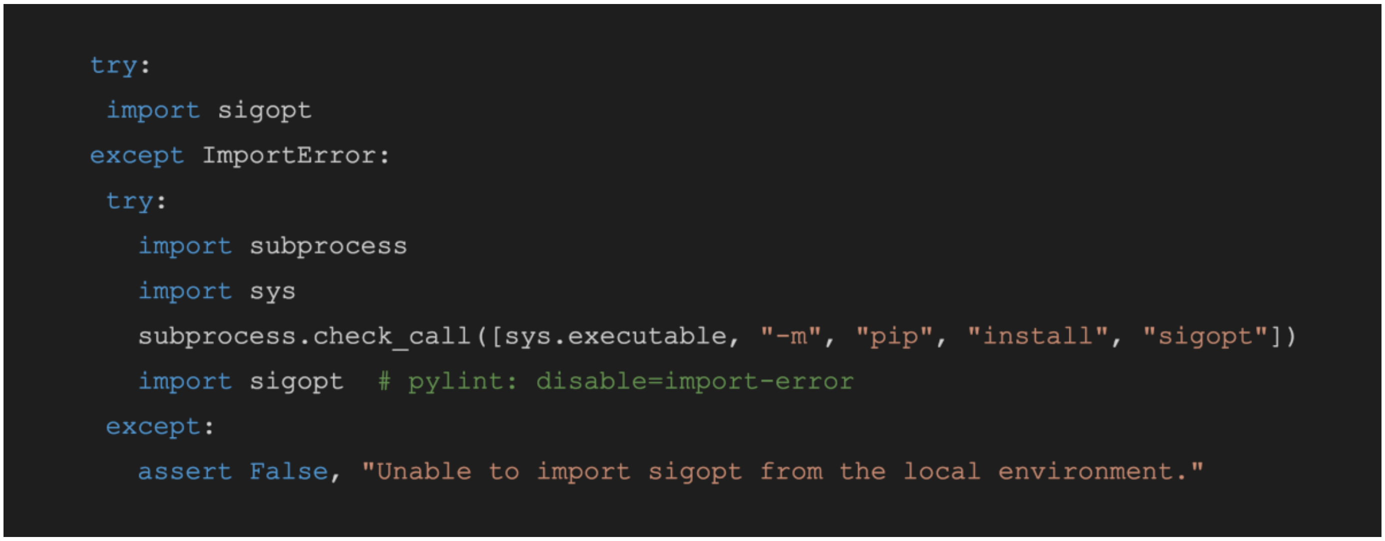
Phantom Dependencies Are No Joke

TL;DR

- For select customers scanned by Endor Labs, the share of Python phantom dependencies in the total of dependencies ranges between 0 and 60%. The share of vulnerabilities in those phantom dependencies in the total of vulnerabilities gets as high as 85%.
- Rebundling is an important problem across ecosystems – thousands of Python and Java components rebundle binary code from other open source projects.

We have coined the term 'phantom dependency' to denote dependencies that exist in an application's code search path – e.g. Java class path or Python library search path, but has not been established through the specific manifest file of the respective application.

The dynamic installation of Python packages at application runtime is one such reason for the presence of phantom dependencies: Originally observed in malicious packages, which use it to make sure that dependencies are present without suspicious manifest file entries, we were surprised to also find this technique being used in legitimate packages – luckily a few dozen only. The PyPI package neural-compressor, for example, uses the following snippet to make sure that the package sigopt gets installed when it is needed.



```
try:
 import sigopt
except ImportError:
 try:
   import subprocess
   import sys
   subprocess.check_call([sys.executable, "-m", "pip", "install", "sigopt"])
   import sigopt  # pylint: disable=import-error
 except:
   assert False, "Unable to import sigopt from the local environment."
```

Other possible reasons for phantom dependencies are manual and scripted installs, bloated container images or Python monorepos, but the impact on SCA tools remains the same: The analysis of a manifest file alone is not sufficient to get an accurate view on the dependencies of an application.

If SCA tools solely rely on the dependencies established through a manifest file, they risk ignoring other dependencies installed in the search path, which can lead to false negatives. If they require to create a lockfile of the entire search path, e.g. using pip freeze, they not only loose the notion of direct vs. transitive dependency, but also take the risk of including too many dependencies, e.g. if the search path includes dependencies from multiple applications or services, as in the case of Python monorepos.

To uncover phantom dependencies in Python, we analyze both first and third party code in Python's search path, and recursively follow import statements in order to create a dependency graph with all the libraries that can be linked to the first party code. This effectively avoids bloated dependency sets – compared to running "pip freeze" on shared environments – and at the same catches dependencies established by other means than the application's manifest file. In the specific case of dynamic installs, however, this also requires that the respective code has been executed beforehand – to make sure that the component is present in the library search path.

Figure 2 shows the prevalence of Python phantom dependencies for a subset of tenants in our platform. The share of phantom dependencies in the total of dependencies ranges from close to 0% up to 60%, whereas the share of vulnerabilities associated with phantom dependencies goes as high as 85% of the total no. of vulnerabilities.

The discrepancy between different tenants suggests that the respective organizations follow different approaches to managing Python dependencies – which reflects the unfortunate reality of Python dependency management. SCA tools should adapt to those practices rather than the other way round.

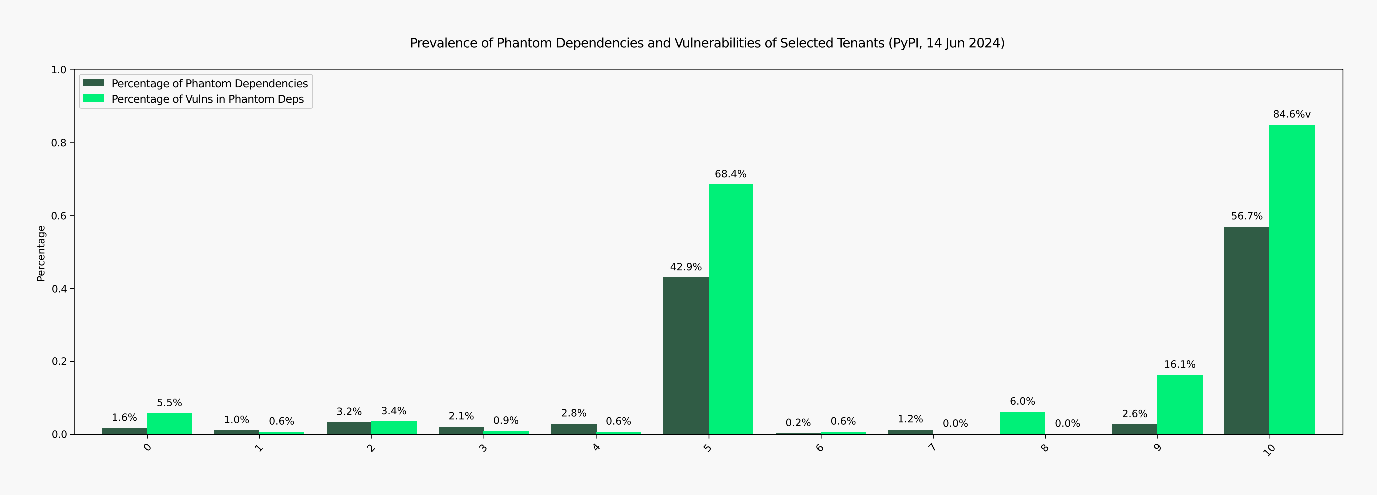


Figure 2: Prevalence of Phantom Dependencies and Vulnerabilities of Selected Tenants (PyPI, 11 Jun 2024)

The Top-20 phantom dependencies across those tenants are as follows. Components like zippor numpy had vulnerabilities in past versions.

1. setuptools
2. enum34
3. pygments
4. zope-event
5. zope-interface
6. hypothesis
7. numpy
8. colorama
9. pytz
10. cffi
11. pycparser
12. python-dateutil
13. ptyprocess
14. pexpect
15. typing-extensions
16. zipp
17. importlib-metadata
18. six
19. iniconfig
20. ply

Another difficulty to capture all dependencies of an application are techniques like rebundling or repackaging, which embed the code of one open source project in the build artifacts of another project – and which is common practice across multiple ecosystems.

In Python for example, it is common practice to include shared binaries, which is supported by a number of bundlers. Binaries like libgcc or liblzma, the latter being produced by the XZ Utils project and subject to a sophisticated supply chain attack in 2024, are contained in the artifacts of hundreds of Python projects. The inclusion of such binaries, however, is not documented in any package metadata, which makes it difficult for SCA tools to spot them. In fact, binary files represent the "vast majority of the content on PyPI, accounting for nearly 75% of the uncompressed size".

A similar problem also exists in other ecosystems: Certain kinds of Java archives, sometimes called Uber JARs, include the Java byte code of other open source projects, e.g. to create self-contained, executable JARs or to avoid package version conflicts. The rebundling or repackaging of Java code is facilitated by multiple build plugins and a common phenomenon. Dann et al., for example, report that 254 known-vulnerable classes from 38 Java components have been repackaged in thousands of other Java components (and tens of thousands of versions) [3]. Sometimes, the original component's metadata is included. In other cases, the metadata is omitted, or the Java byte code has been changed due to recompilation or change of package namespaces.

	Recompiled	**Uber-JAR**	**Uber-JAR (w/o meta)**	**Repackaged**
# rebundled classes	143 / 254	222 / 254	222 / 254	17 / 254
# distinct component versions	5,919	36,609	24,500	168
# distinct components	360	6,728	3,882	89

Part 2: Discrepancies and Shortcomings of Vulnerability Databases

Ecosystem Coverage of CVEs

TL;DR

- 18% of npm vulnerabilities in the public Open Source Vulnerability (OSV) database do not have CVE aliases, thus, npm application developers should make sure that their SCA solution ingests OSV.
- Spotchecks of OSV vulnerabilities without CVE alias illustrate different phenomena, including genuine problems related to bundled binaries in Python projects, or erroneous maintenance of existing CVE aliases.

The recent problems of the NVD raise the question whether vulnerability managers have to consider other, additional databases.

The Open Source Vulnerability database (OSV) established itself as a valuable resource for vulnerabilities in open source projects. It ingests and aggregates several vulnerability databases to cover a wide range of 26 ecosystems — from mainstream to more exotic ecosystems like R or Haskell.

As of June 11th, 2024, the OSV contains a total of 13,963 known vulnerabilities for the prominent ecosystems PyPI, Maven, npm, NuGet, Go and RubyGems.org. 828 of those vulnerabilities do not have a CVE alias, i.e. they would be missed when relying on NVD only, most of them stemming from the GitHub Advisory Database (GHAD).

The next figure illustrates how the number of vulnerabilities without CVE aliases varies per ecosystem – between 1% for Maven and 18% for npm – which suggests that some ecosystems are better covered by NVD than others. The take-away for users of SCA solutions is to check which sources are covered by their respective provider, especially if they develop in JavaScript/npm.

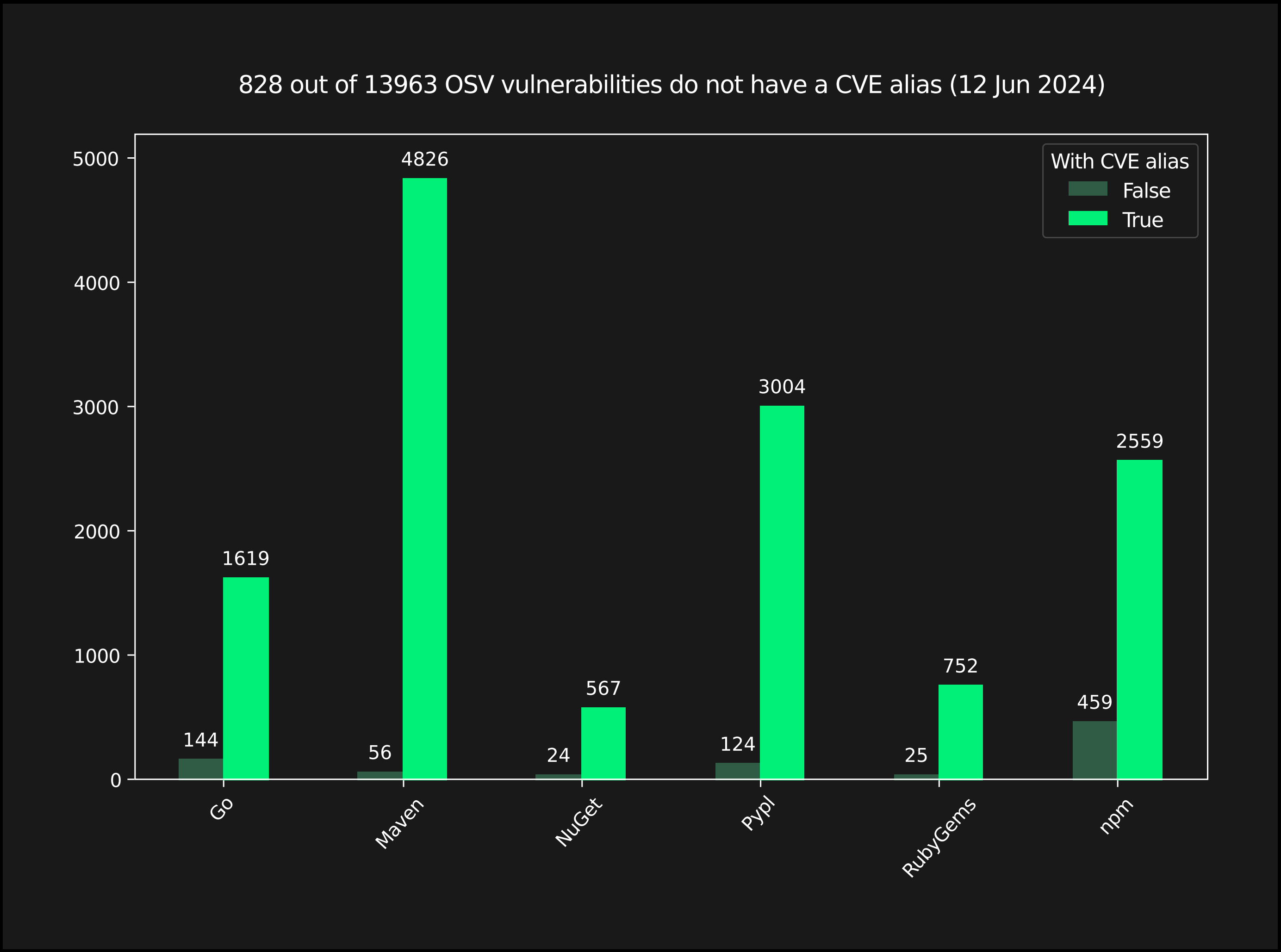


Figure 3

But even where the numbers appear little, it shall be noted that some affect well-known components like PyPI/pillow, PyPI/tensorflow or npm/angular. The next chart displays the Top-50 components that have the most vulnerabilities not aliased by a CVE.

One important question is whether those vulnerabilities matter at all, i.e. whether they are real or disputed, have low or critical severity and whether the affected components are used at all.

A review of all the 11 vulnerabilities for RubyGems/nokogiri shows that they all revolve around the update of bundled binaries, especially libxml2, or vendored code. As such, they overcome the problem discussed in the previous section, which is that SCA solutions struggle to detect bundled libraries and vendored code. Apart from GHSA-r95h-9x8f-r3f7, which clearly states that a vulnerability in libxml2 has no impact on nokogiri, the vulnerabilities seem relevant and should indeed be highlighted to users of nokogiri.

The 9 vulnerabilities for Maven/com.vaadin:vaadin-bom are wrongly reported for not being related to CVEs, due to the fact that the OSV entries do not maintain the alias field. Interestingly, vaadim-bom is a Maven project of type pom, also called BOM POM, and thus will not necessarily appear as a dependency itself. As a consequence, the vulnerabilities associated to vaadim-bom may not be mapped to users of the Vaadim components that actually carry the vulnerable code, which is why we associate them to other Vaadim components, e.g. vaadin-checkbox-flow in case of CVE-2021-33605 (GHSA-hw7r-qrhp-5pff).

The 8 vulnerabilities for PyPI/pillow are a mix of different phenomena discussed before: PYSEC-2023-175 and GHSA-56pw-mpj4-fxww address the bundling of a vulnerable version of the libwebp binary (without referring to each other through an alias). GHSA-jgpv-4h4c-xhw3 is actually related to a CVE, but the entry is not linked through an alias. OSV-2022-1074 and OSV-2022-715 have been automatically created as part of the OSS-Fuzz project, without any impact and severity being given, and the other three refer to genuine vulnerabilities within Pillow itself.

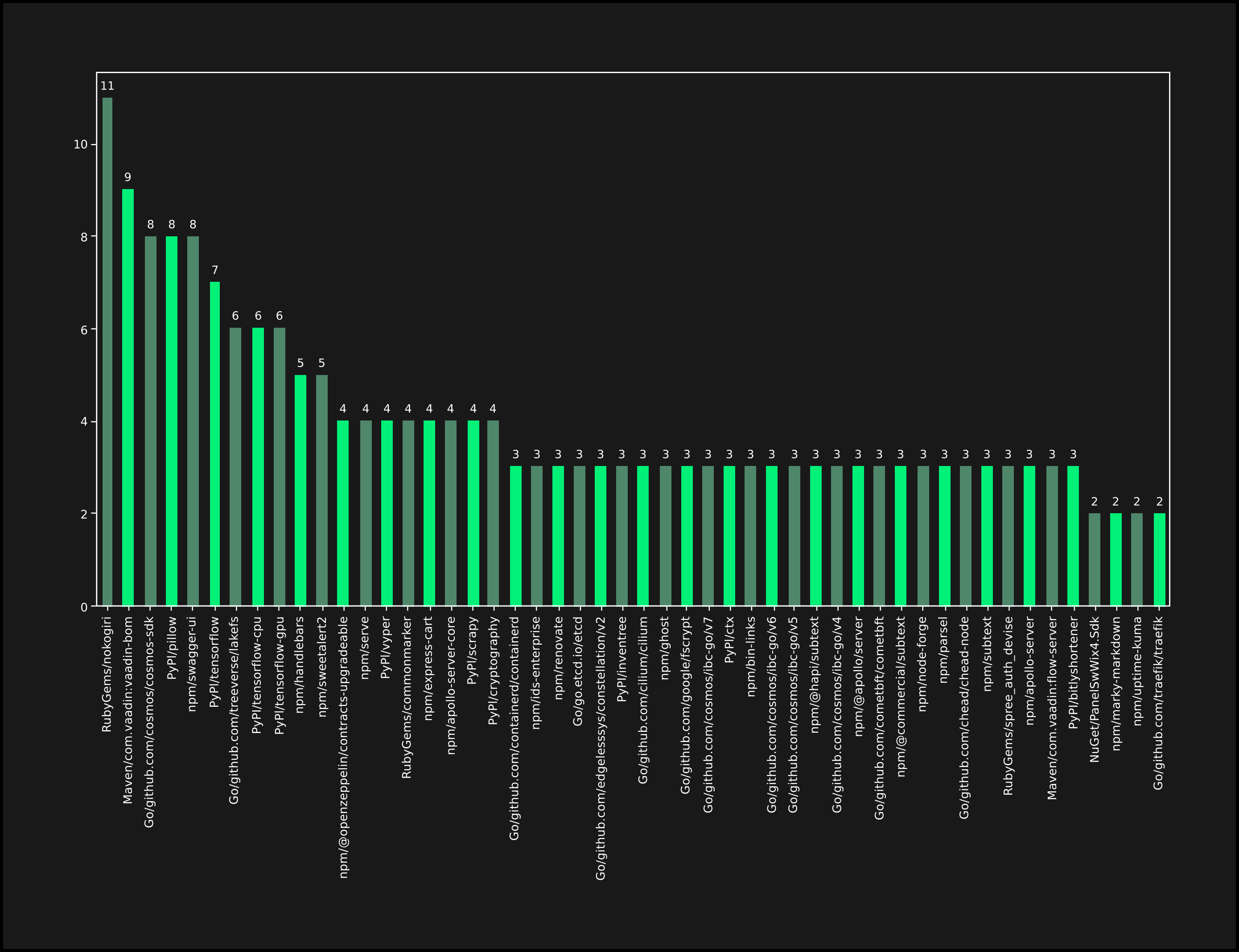


Figure 4: Number of OSV vulnerabilities without CVE aliases per package (04 Jun 2024)

Vulnerability Advisory Publication Delays

TL;DR

- 2,527 out of 3,655 security advisories (69%) are published after the corresponding security release, with a median delay of 25 days, which increases the window of opportunity for attackers to exploit vulnerable systems.

- For 25% of 5,978 vulnerabilities having both a CVE and a GHSA, the GHSA is published 10 or more days after the CVE. During this delay, organizations depending on those sources rely on imprecise CPE-mapping to identify vulnerable components in their stack.

Another important aspect in regards to vulnerability management and vulnerability databases is the timely fix by project maintainers, the publication of corresponding security releases - which contain the respective code changes, and the publication of security advisories that inform OSS users about the presence of vulnerabilities and the availability of available security releases.

The figure below presents relevant events in the timeline of security fix propagation and advisory publication, and various delays that hinder the timely notification of open source users, and increase the window of opportunity for attackers. Minimizing those delays is paramount considering the speedy development of exploits, and the fact that additional delays will be introduced through organizations' patch processes.

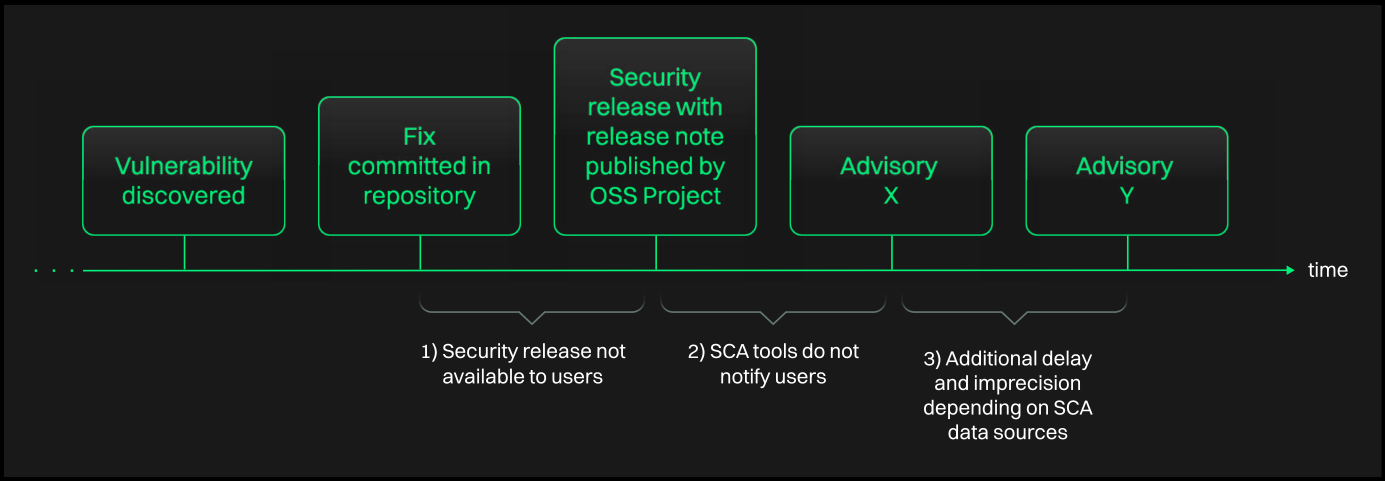

Vulnerability discovered
Fix committed in repository
Security release with release note published by OSS Project
Advisory X
Advisory Y
time
1) Security release not available to users
2) SCA tools do not notify users
3) Additional delay and imprecision depending on SCA data sources

Figure 5: Timeline of security fix propagation and possible delays

The first potential delay happens between the fix of a vulnerability in a project's source code repository and the publication of a corresponding security release in a public repository such as Maven or npm, which facilitates the consumption of the fix by downstream users. This delay matters, because attackers may be able to spot the fix commits, hence, know about technicalities of the vulnerability and how to exploit it, while users have no easy way to patch their systems.

Researchers from the North Carolina State University (NCSU) found "that open source packages include security fixes in a new release within 4 days of the fix at the median, npm being the fastest and Maven being the slowest. However, 25% of the releases in our data set still came at least 20 days after the corresponding security fix".

The window of opportunity may be further increased if a security advisory is published after the security fix release. This has been the case for 2,527 advisories, which corresponds to 69.1% of NCSU's dataset, and the following table shows the median delay for individual ecosystems, and differentiating between CVEs and non-CVEs.

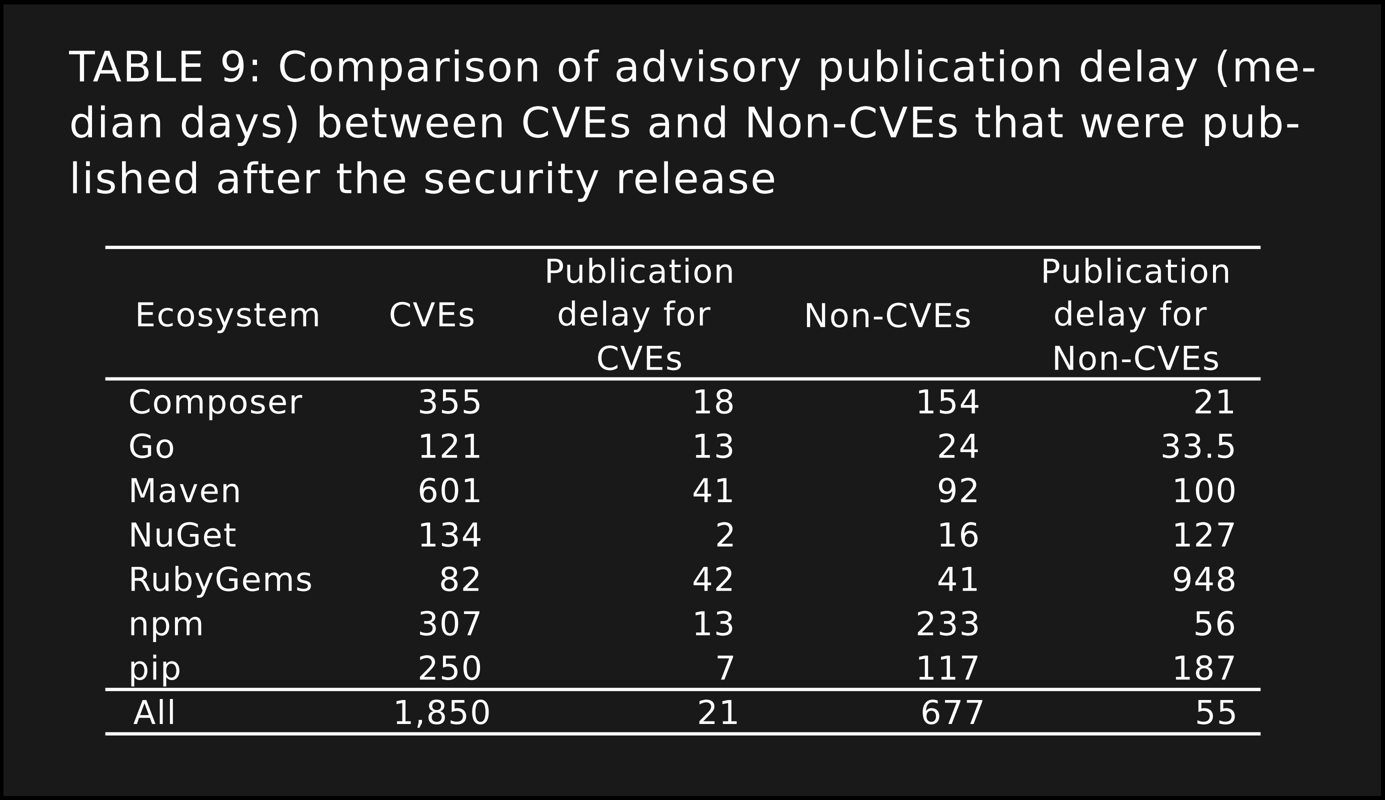


TABLE 9: Comparison of advisory publication delay (median days) between CVEs and Non-CVEs that were published after the security release

Ecosystem	CVEs	Publication delay for CVEs	Non-CVEs	Publication delay for Non-CVEs
Composer	355	18	154	21
Go	121	13	24	33.5
Maven	601	41	92	100
NuGet	134	2	16	127
RubyGems	82	42	41	948
npm	307	13	233	56
pip	250	7	117	187
All	1,850	21	677	55

But the publication of a security advisory or vulnerability information can take different forms – from blog posts and CVEs to GitHub Security Advisories (GHSAs), and whether your SCA tool learns about the advisory depends on its specific information sources.

When comparing the publication dates of CVEs with the sources referenced by those CVEs, a previous study by Anwar et al. showed that ≈ 28% of CVEs have a publication lag of more than a week. In other words, they lag more than one week behind the publication of the same vulnerability in another information source such as “vulnerability databases (e.g. Security Focus), bug reports or email archives threads (e.g. Bugzilla), or security advisories (e.g. cisco.com)”.

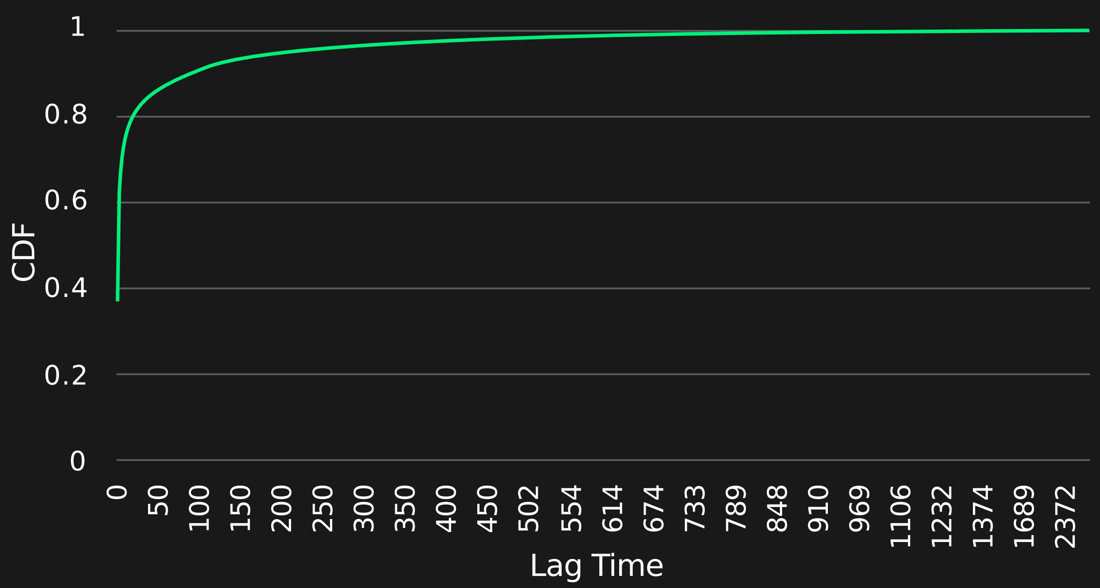


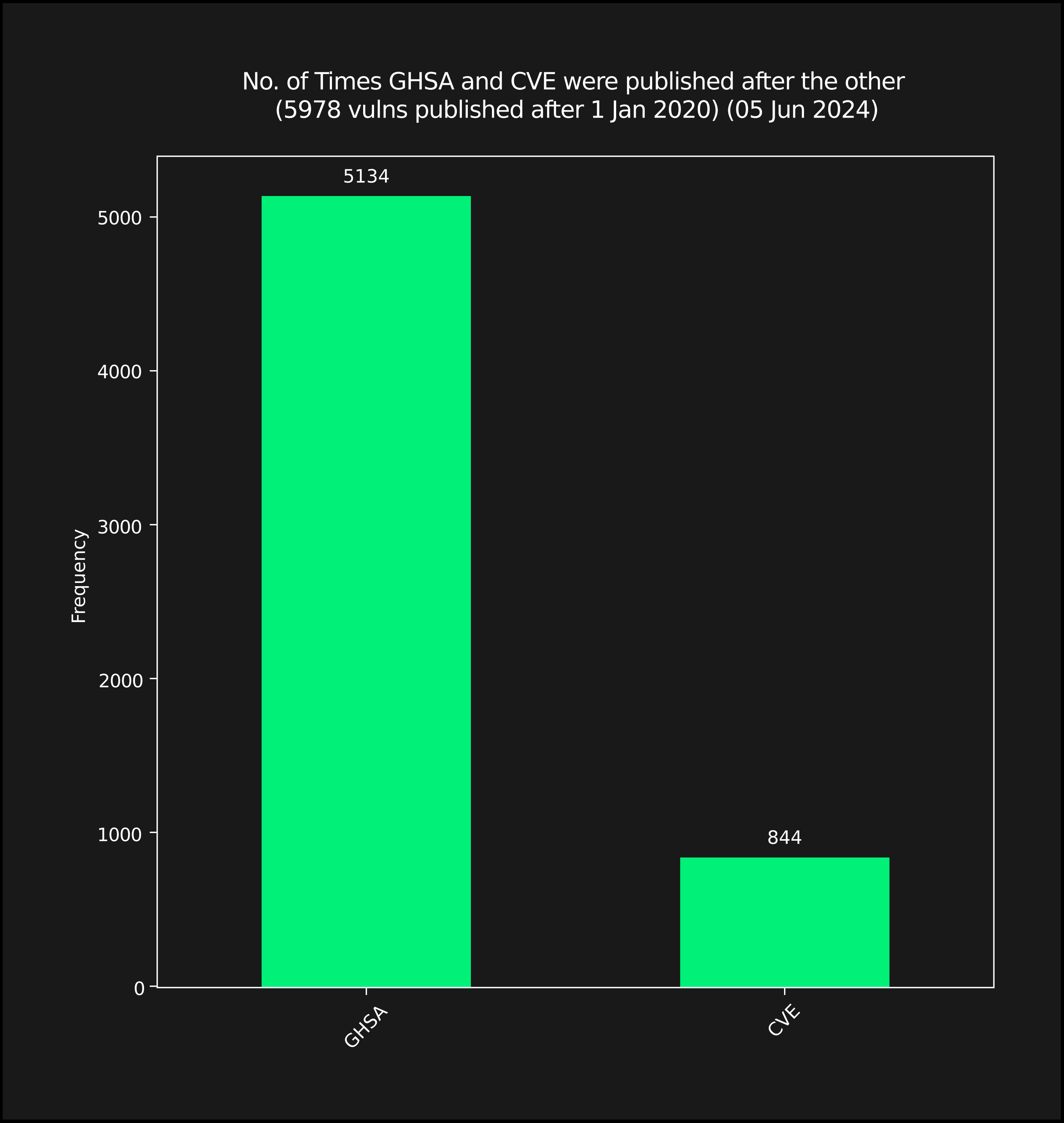
Figure 1: CDF of vulnerability lag times. Lag time is the number of days after our estimated disclosure date when a vulnerability enters into the NVD. Note, 38% of the vulnerabilities have no lag.

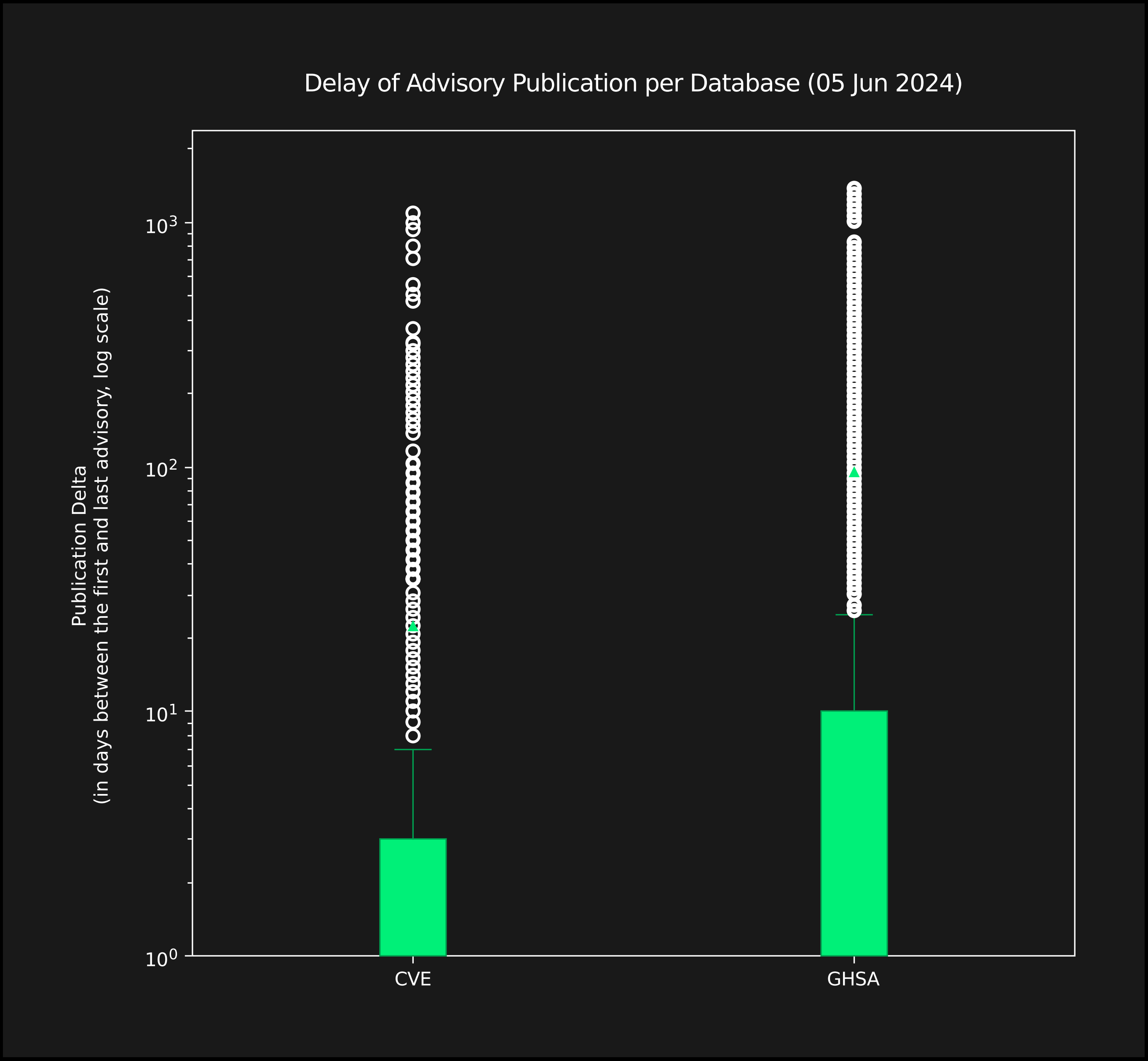
For this report, we specifically compare the publication dates of vulnerabilities published both as GHSA in the GitHub Advisory Database and as CVE in the NVD. This is important for two reasons: First, because consumers of GHSAs benefit from the use of package identifiers that match the respective ecosystems, whereas the CPE identifiers used by CVEs only support imprecise mappings. Second, because organizations that only consume NVD data feeds may suffer from a publication delay comparable to what has been observed in the study mentioned above.

5,978 vulnerabilities aggregated by OSV have exactly one GHSA and one CVE as alias. The large majority of those vulnerabilities are first published in the NVD, 25% of those cases with a delay of more than 10 days. This means that the accurate mapping of vulnerabilities using ecosystem-specific package identifiers is significantly delayed, i.e. SCA tools solely relying on those sources can only use the imprecise CPE mapping during this lag.

Conversely, less than 1K vulnerabilities are first published as GHSA, with 75% of CVEs being delayed by less than 3 days.

[2] Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages





Public Advisories Lack Code-level Information

TL;DR

- Across six ecosystems, 47% of advisories in public vulnerability databases do not contain any code-level vulnerability information at all.
- 51% contain one or more references to fix commits, and only 2% contain information about affected functions.

The application of program analysis techniques requires **code-level information about vulnerabilities**, e.g. the names of affected functions or the fix commits that were developed by OSS project maintainers to overcome a vulnerability. Without such information, it is not possible to establish whether known-vulnerable functions can be executed in the context of a downstream application.

Even though the prose CVE descriptions of some open source vulnerabilities mention the names of affected functions or classes, this information is often incomplete and unstructured, and thus can hardly be used as reliable input for program analysis.

But what are the consequences of this lack of code-level information in public vulnerability databases? First, commercial SCA providers build proprietary databases to complement public advisories. The significant costs of building those make them – understandably – reluctant to share this information, which in turn makes it impossible for end-users and open source projects to validate, and compare it across vendors. Second, it is difficult for open source projects to implement effective code-centric SCA solutions.

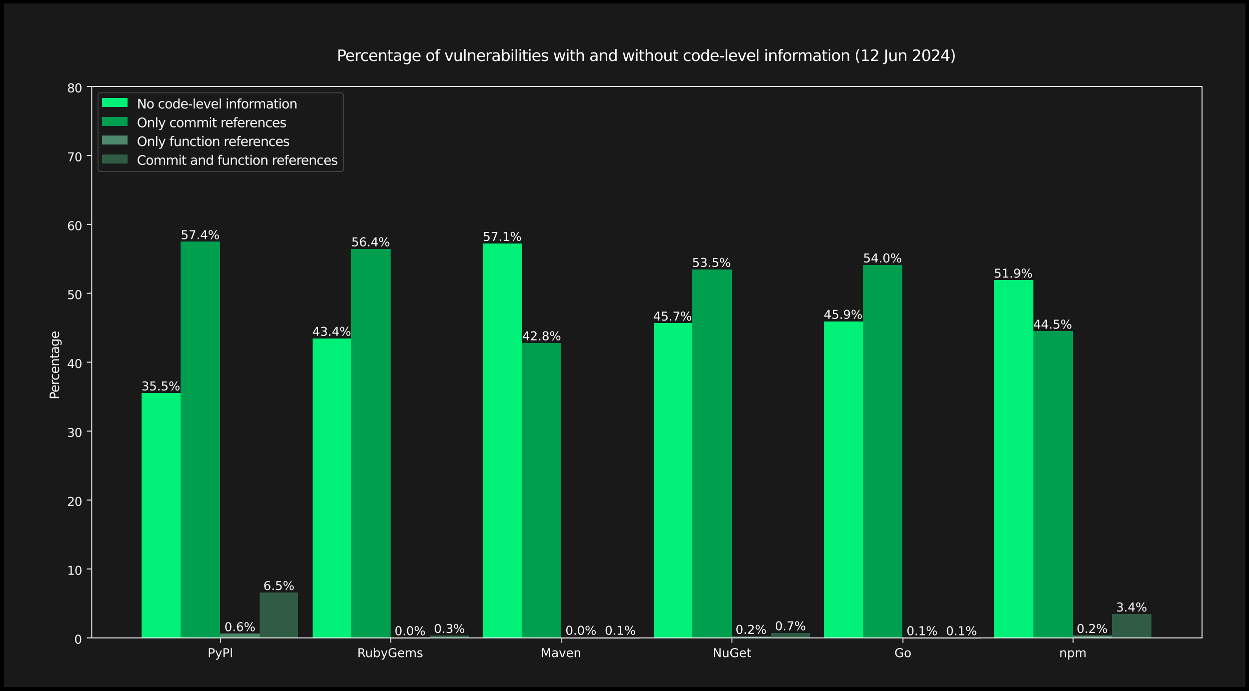
The advent of the OSV database with its additional data sources compared to NVD, and the sporadic emergence of security advisories with structured information about affected functions motivated us to investigate the current state of affairs with regard to the availability of code-level vulnerability information.

As before, we grouped all OSV advisories linked through aliases and considered them as one vulnerability. We then checked whether any advisory for a given group contains a reference to a fix commit, names of affected functions, both of it or none of it. Positive examples are the vulnerabilities (GHSA-hm9v-vj3r-r55m, CVE-2023-36807) for Python/PyPI and (GHSA-35jh-r3h4-6jhm, CVE-2021-23337) for JavaScript/npm, where the GHSA complements the CVE information with fix commits and vulnerable function identifiers.

However, the following chart shows that code-level information is still not widely available in public advisories. The majority of Java/Maven and JavaScript/npm vulnerabilities do not have any code-level information at all (57% and 52%). And even where fix commits are available, e.g. in the majority of Python/PyPI and C#/Nuget vulnerabilities (57% and 54%), it is not clear whether those fix commits are complete (i.e. cover all maintained release branches), or whether they contain changes unrelated to the vulnerability fix.

The number of advisories with affected function identifiers is insignificant except for Python/PyPI and JavaScript/npm, where it reaches 7% and 4% respectively.

[1] https://ieeexplore.ieee.org/iel7/8013/5210089/10381645.pdf



Affected components and component versions

TL;DR

- Across 4 ecosystems, Endor Labs and OSV only agree in 72% cases on the affected packages. In 12% of the cases, they mark an intersecting set of components as affected, and in 10% of the cases, Endor Labs marks one additional package.

Naming the components affected by a given vulnerability is more difficult than it seems, due to various reasons. Examples are open source projects that produce multiple artifacts, project forks or artifact identifiers that change over time or depending on the distribution channel.

CVE-2023-41080, for example, affects Apache Tomcat. The particularity of this project is that Tomcat exists for many years already, that its Maven coordinates changed multiple times and that the same classes are contained in multiple artifacts, e.g. the stand-alone version of the Tomcat Web server and its embedded version.

In this particular case, the vulnerable class FormAuthenticator is contained in the following project artifacts, which exist for the major Tomcat releases 7 to 11: org.apache.tomcat:tomcat, org.apache.tomcat:tomcat-catalina and org.apache.tomcat.embed:tomcat-embed-core.

Previous Tomcat releases, however, were distributed with different Maven coordinates, namely tomcat:catalina (for major versions up to 5, with the latest release from 2007) and org.apache.tomcat:catalina (for major version 6, with the latest release from 2017), both of which also contain the affected class. For users of those old versions, which are not explicitly marked as vulnerable neither by NVD nor in the Apache mailing list, the big question is whether they are safe…

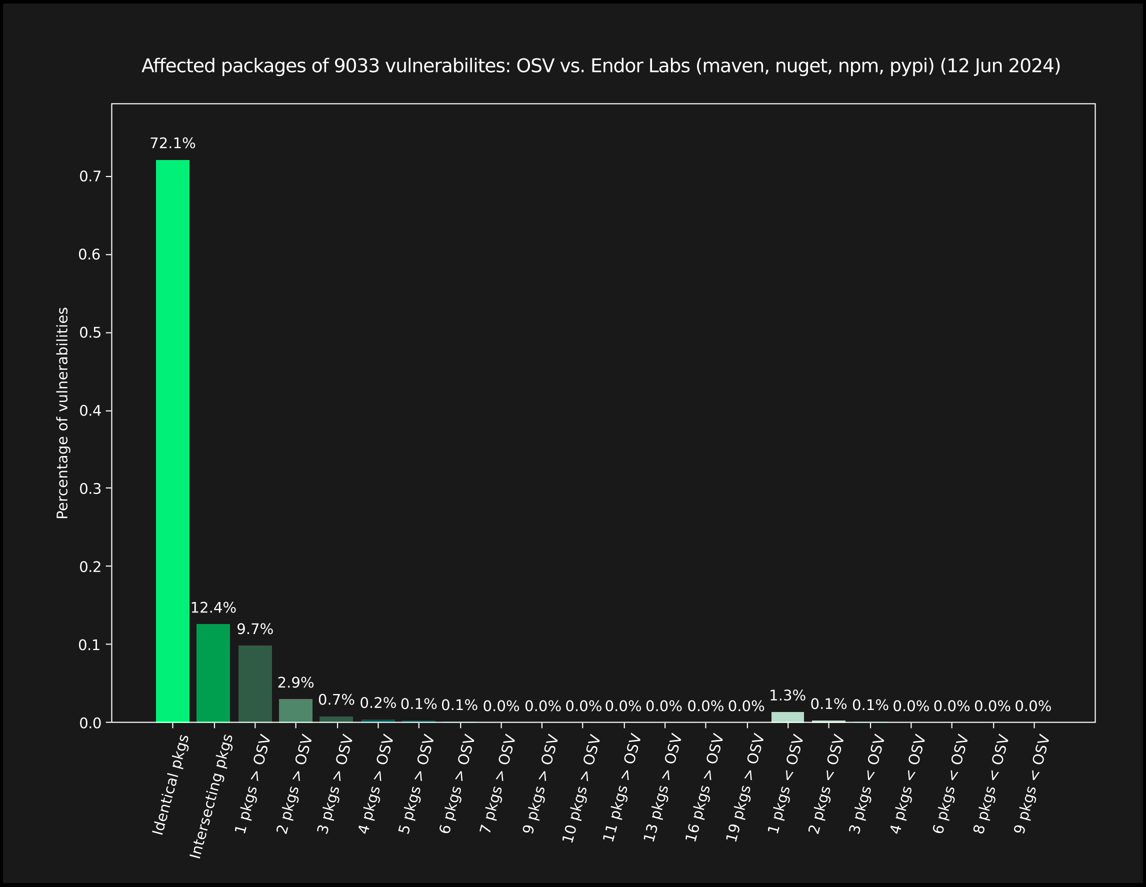
Other examples comprise BOM dependencies or parent POMs in the Maven ecosystem. Both can be used to harmonize dependency versions, however, they typically do not contain any code itself and may not show up in the dependency graphs of downstream users. Examples are the vaadim-bom discussed in a previous section, or CVE-2017-12159 (GHSA-7fmw-85qm-h22p), which refers to maven/org.keycloak:keycloak-parent, while the vulnerable code is contained in maven/org.keycloak:keycloak-services.

Another interesting case is CVE-2023-36566, affecting the Microsoft Common Data Model SDK: GHSA-vm2m-7hpw-fpmq mentions affected components only for the Maven, NuGet and PyPI ecosystems, but the original advisory from Microsoft also mentions an npm package, which is produced from the same GitHub source code repository.

CVE-2024-35220 (GHSA-pj27-2xvp-4qxg) illustrates the problem of project forks: Originally published as npm/fastify-session, the project is now continued in another GitHub repository and published as npm/@fastify/session. While the GitHub advisory only mentioned the latter as affected, we also added the former as affected, which allows alerting customers who have not migrated yet.

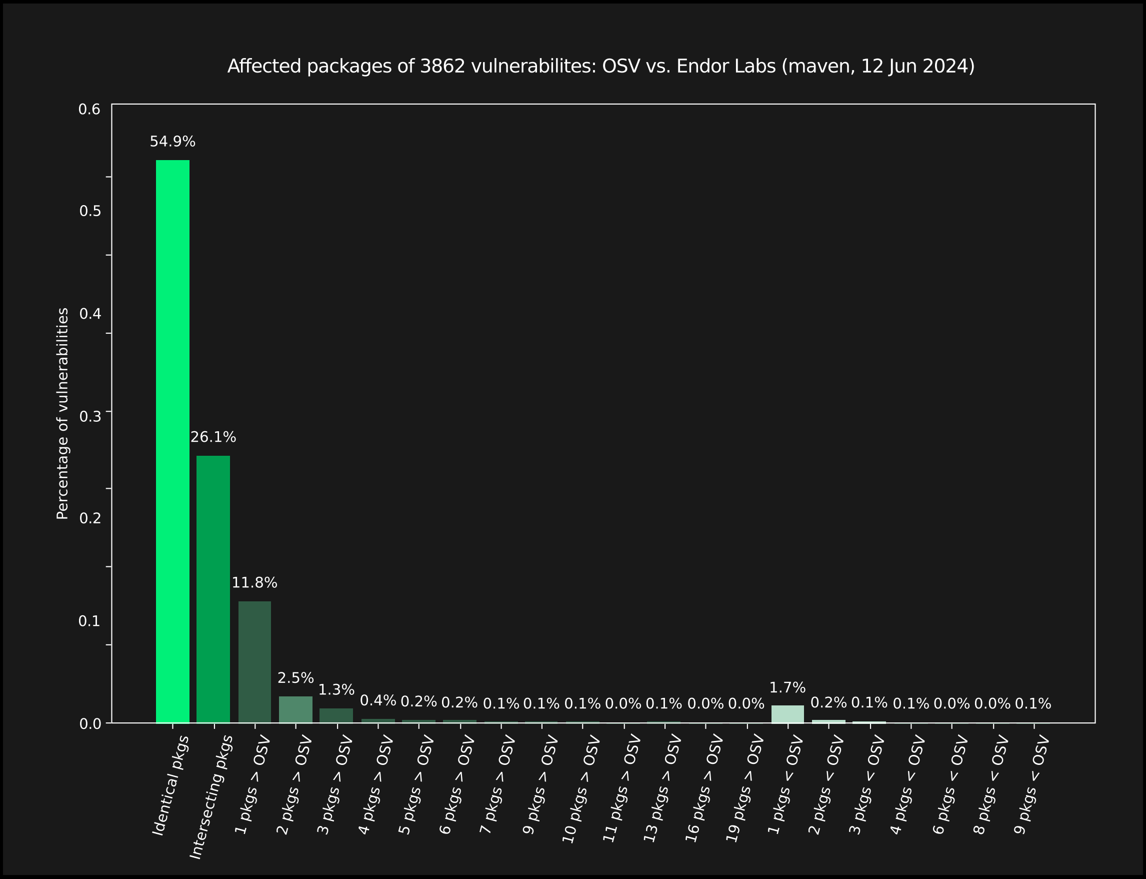
To summarize, the following figures illustrate to what extent OSV and Endor Labs deviate in regards to the affected components.

Across four ecosystems, they agree in 72% of the vulnerabilities on the affected component identifiers. In 10% of vulnerabilities, Endor Labs marks one additional component as affected. In 1% of vulnerabilities, OSV marks one additional package as affected. The example above, CVE-2017-12159 affecting Keycloak, is part of the “Intersecting pkgs”, where OSV mentions a component that Endor does not and vice-versa.



Affected packages of 9033 vulnerabilites: OSV vs. Endor Labs (maven, nuget, npm, pypi) (12 Jun 2024)

When looking at the Maven ecosystem alone, the discrepancy gets more accentuated: OSV and Endor only agree in 55% of vulnerabilities on the affected components.



Affected packages of 3862 vulnerabilites: OSV vs. Endor Labs (maven, 12 Jun 2024)

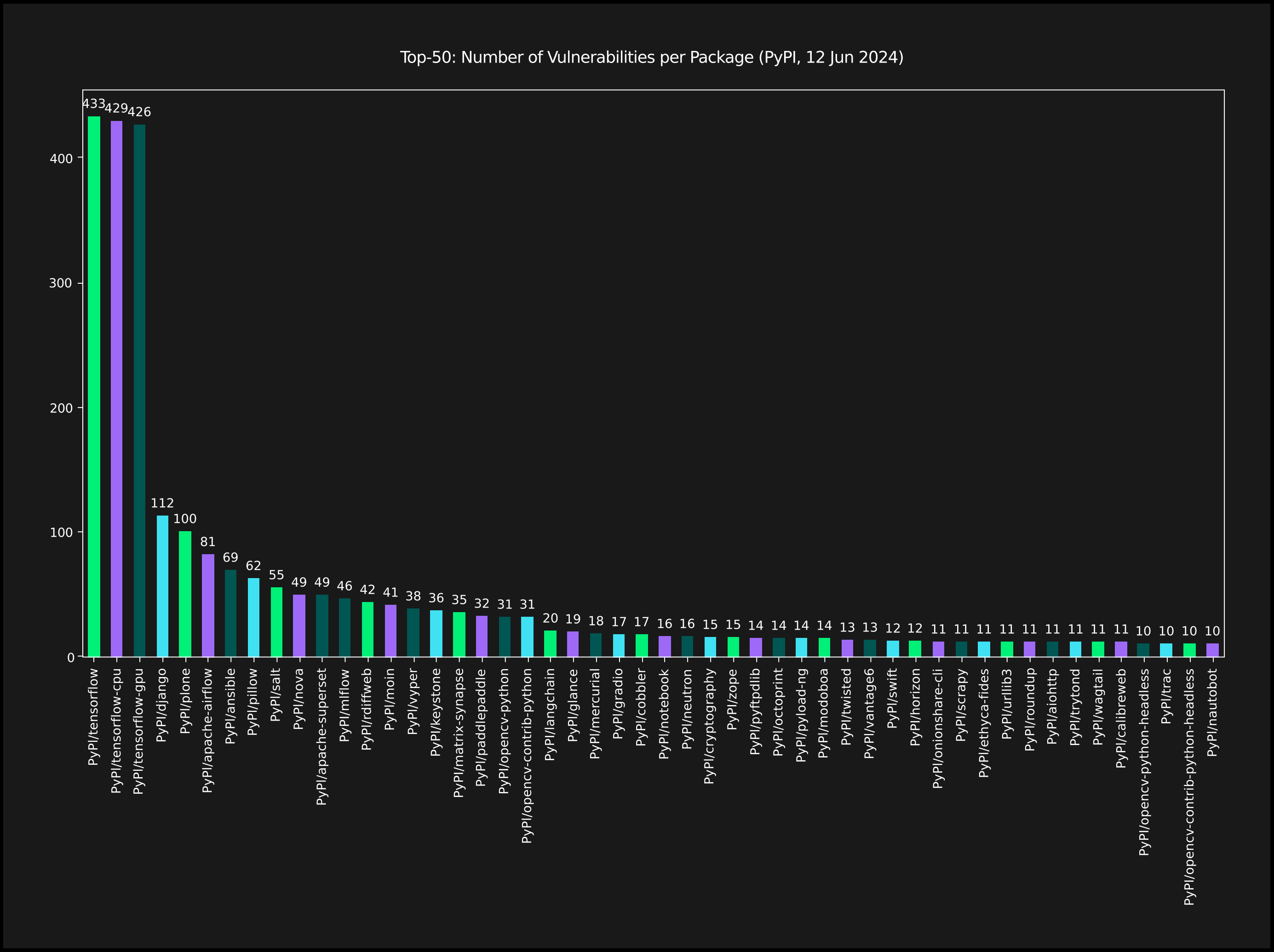
Notable Impacts of Mature Vulnerability Prioritization Practices

TL;DR

- Prioritization lets organizations focus on less than 5% of their total number of vulnerabilities.
- In the case of the Python ecosystem, updating the Top 20 components to non-vulnerable versions would remove more than 75% of all the vulnerability findings (Java 60%, npm 44%).

The first figure shows the Top-50 components having the most known vulnerabilities in OSV across six ecosystems (npm, Maven, PyPI, Go, RubyGems and NuGet).

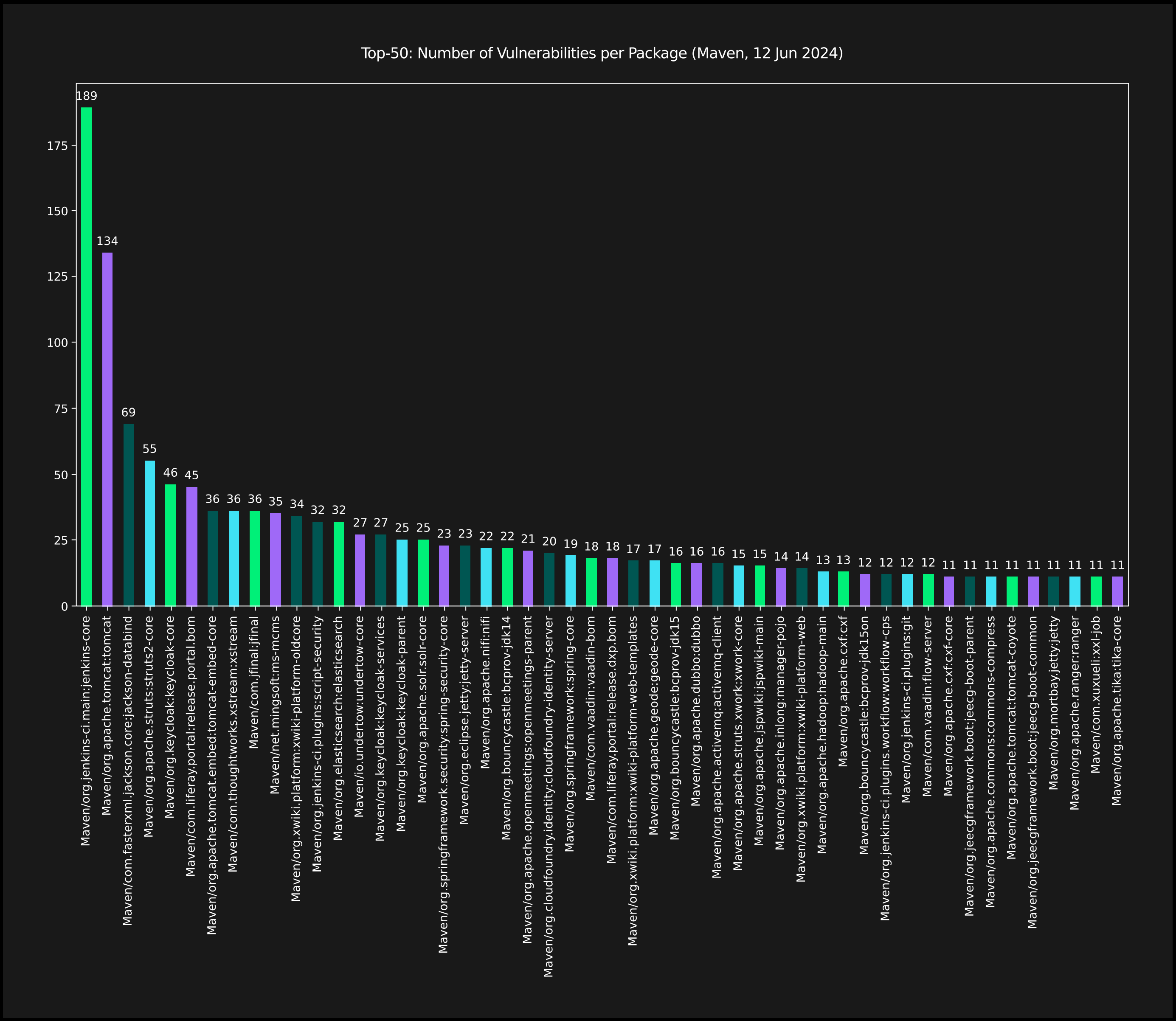
The prevalence of Python packages, with Tensorflow having the most known vulnerabilities, underlines the importance of handling Python dependencies beyond what's mentioned in manifest files. By following the import statements, for example, it is possible to get a better understanding of the effective dependency tree, compared to relying on the flattened output of "pip freeze", which may be hugely inflated if multiple Python projects load dependencies from the same Python library search path.

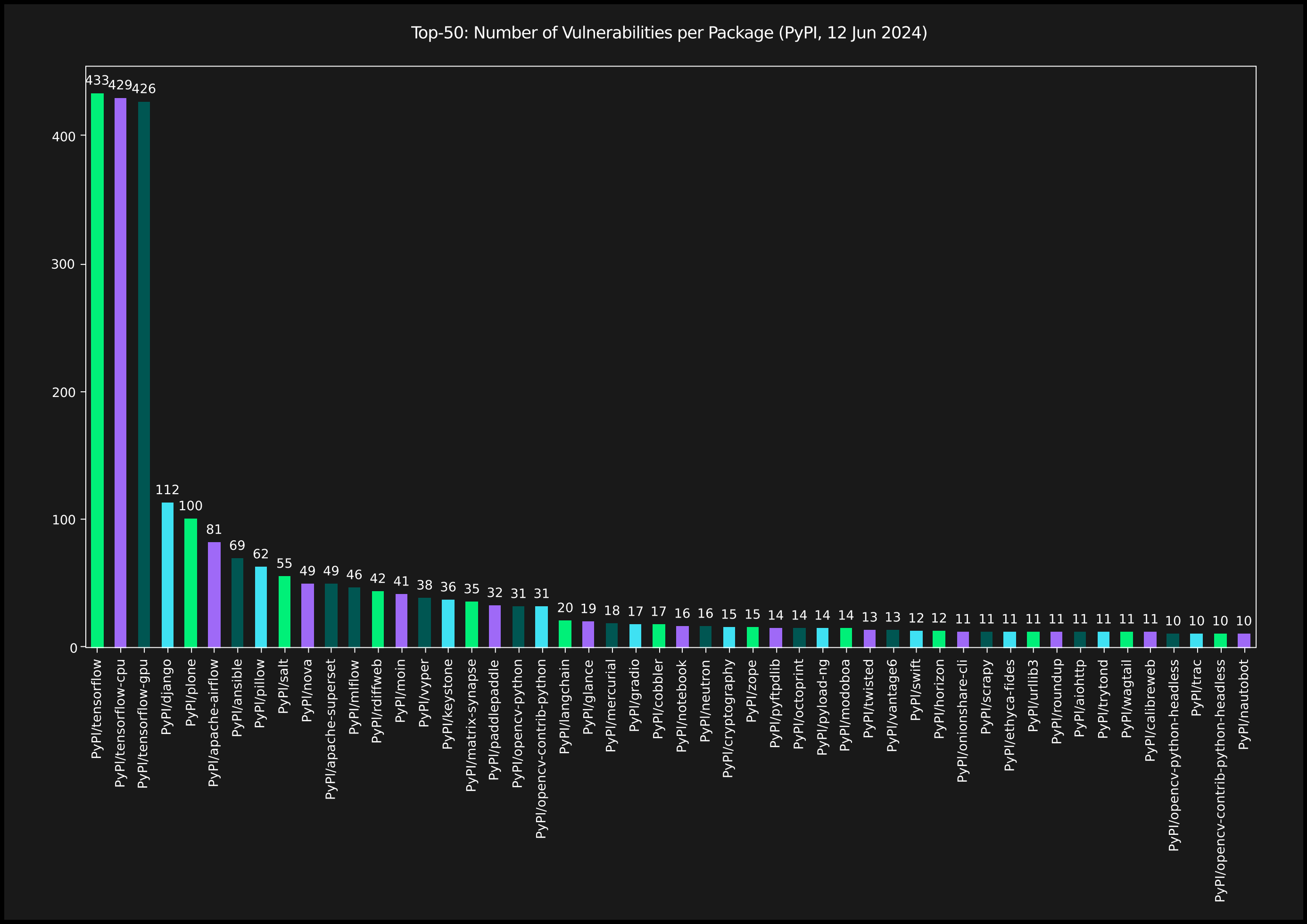


The next figures show the Top-50 Java/Maven and Python/PyPI packages with the most known vulnerabilities.

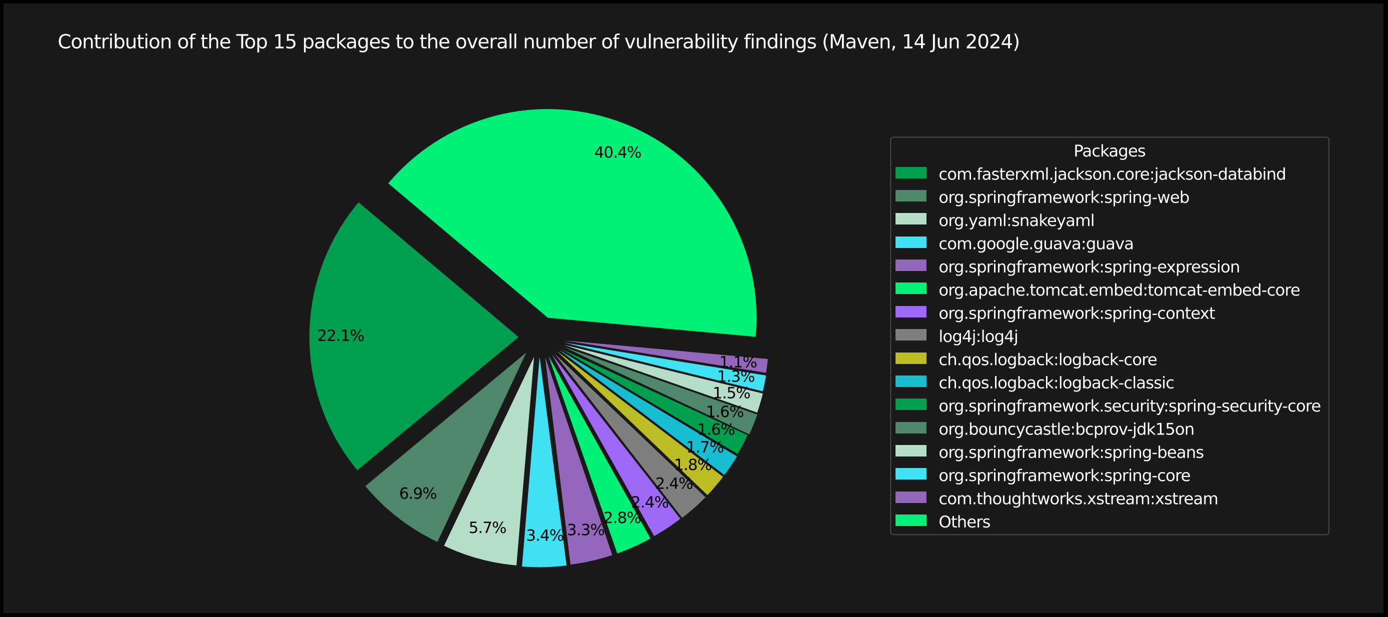
Jenkins is stand-alone software, and should be considered apart. Apache Tomcat can be run stand-alone or embedded, e.g. in self-contained Spring Boot applications. Here, the difference between org.apache.tomcat:tomcat and org.apache.tomact.embed:tomcat-embed-core is striking, and the question arises whether the advisories are complete.

One such example is CVE-2020-13935, which is caused by invalid payload lengths in WebSocket frames. According to OSV, the vulnerability affects org.apache.tomcat:tomcat. According to our analysis, however, the vulnerable classes are comprised in five different Tomcat artifacts, including org.apache.tomcat.embed:tomcat-embed-core or org.apache.tomcat.embed:tomcat-embed-websocket, and exploit PoCs showed that they are indeed exploitable.

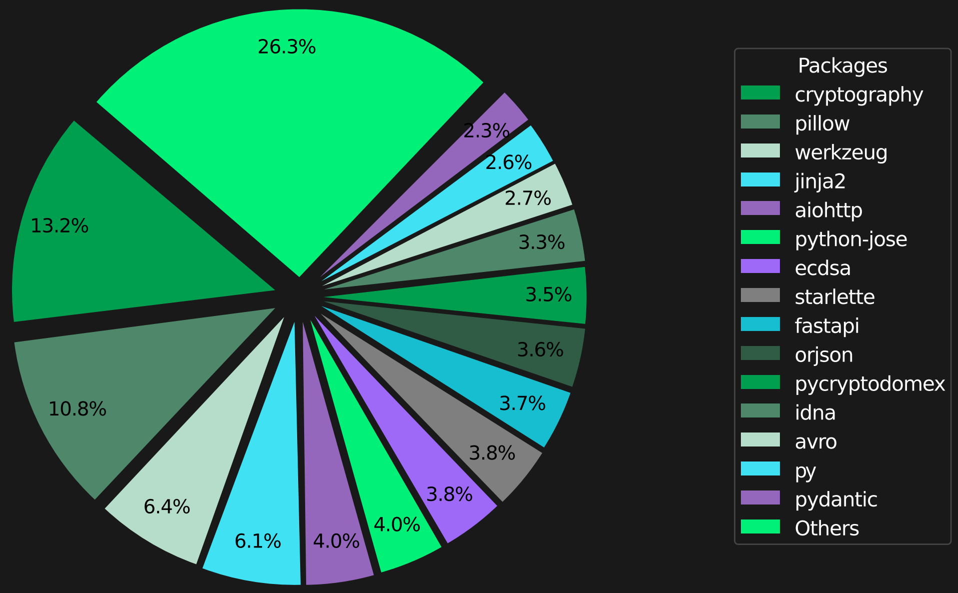




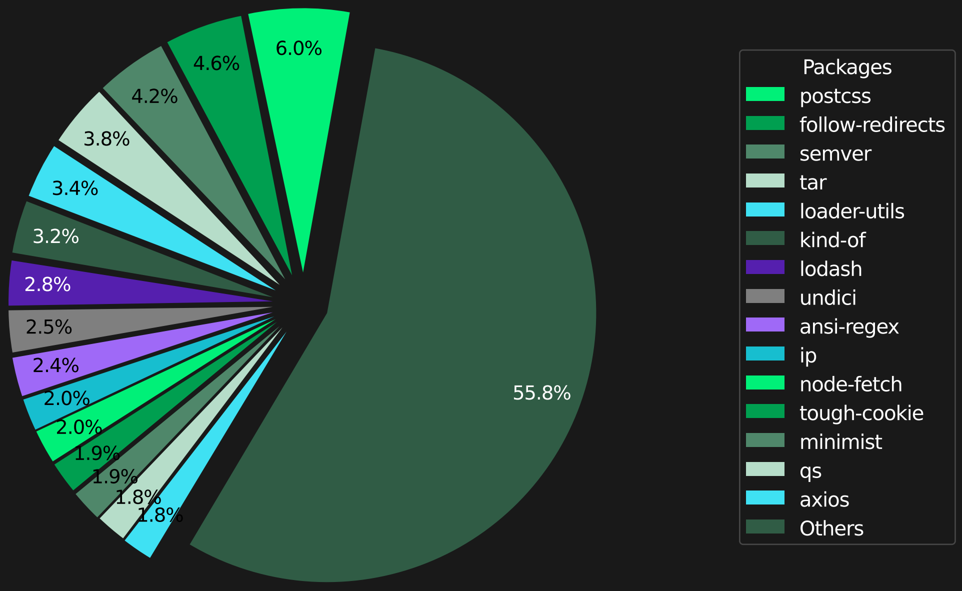
The three pie charts shown below show the Top 15 components bringing in the most findings across our customer base. In the case of the Python ecosystem, updating the Top 15 components to non-vulnerable versions would remove close to 74% of all the vulnerability findings (Java 60%, npm 45%).



Contribution of the Top 15 packages to the overall number of vulnerability findings (Pypi, 14 Jun 2024)

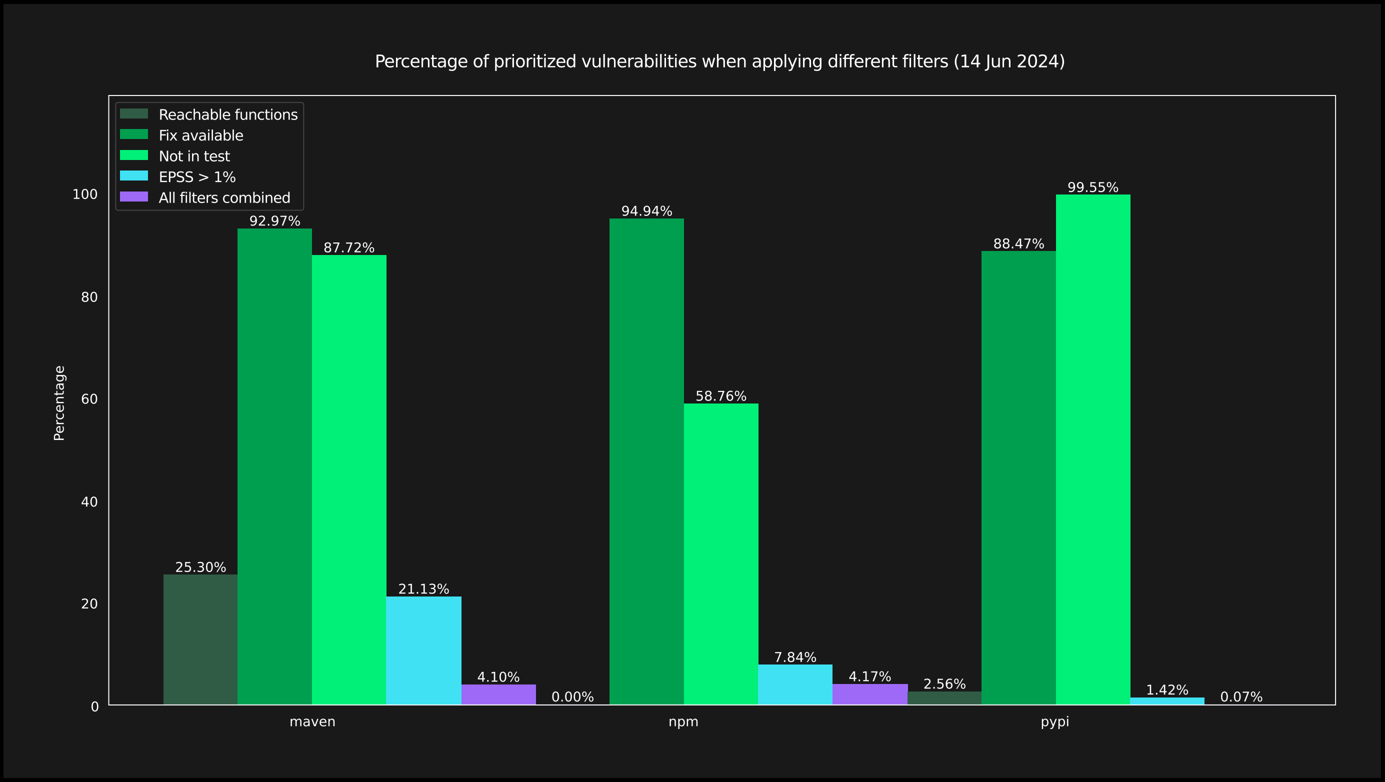


Contribution of the Top 15 packages to the overall number of vulnerability findings (Npm, 14 Jun 2024)



The next chart shows the percentage of vulnerabilities that can be prioritized when applying different filters. The most effective individual filters only consider vulnerabilities whose vulnerable code is reachable, and whose EPSS score is greater than 1%.

When combining all filters, organizations can focus on roughly 4% of Java and JavaScript vulnerabilities (note that function-level reachability is not yet available for JavaScript), and less than 1% of Python vulnerabilities.



Vulnerabilities in AI Projects

TL;DR

- For tensorflow, the number of vulnerabilities reported by GHSA vs. other vulnerability databases differs by more than 40 (~10%).
- 4 ML/AI projects are entirely missed by GHAD (pandas, dask, numexpr, pytorch-lightning), while they have vulnerabilities reported in other databases.

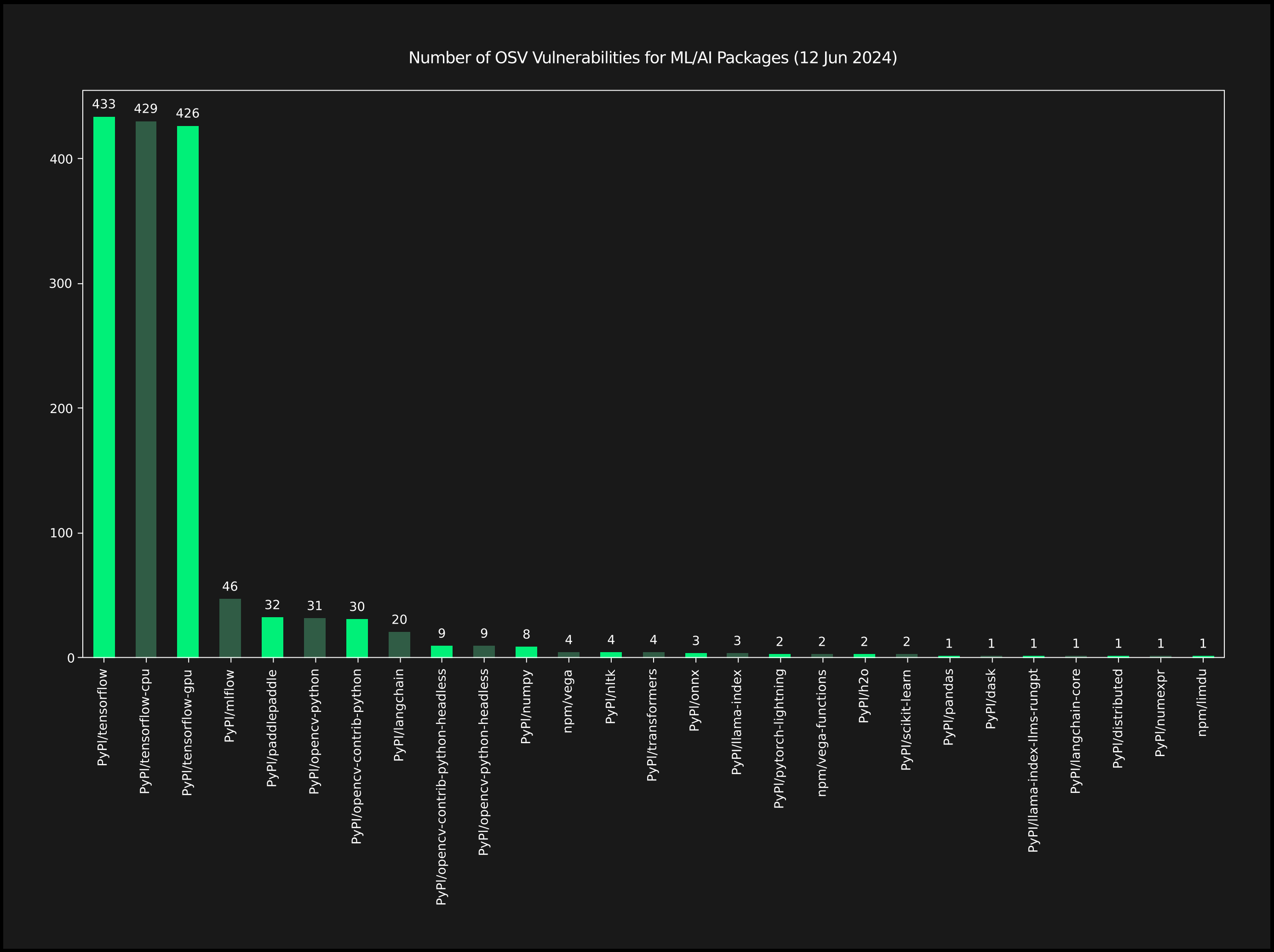
Above discussions regarding vulnerability databases apply across the entire open source ecosystem. Here we have a closer look at open source components that are of particular interest for ML/AI applications.

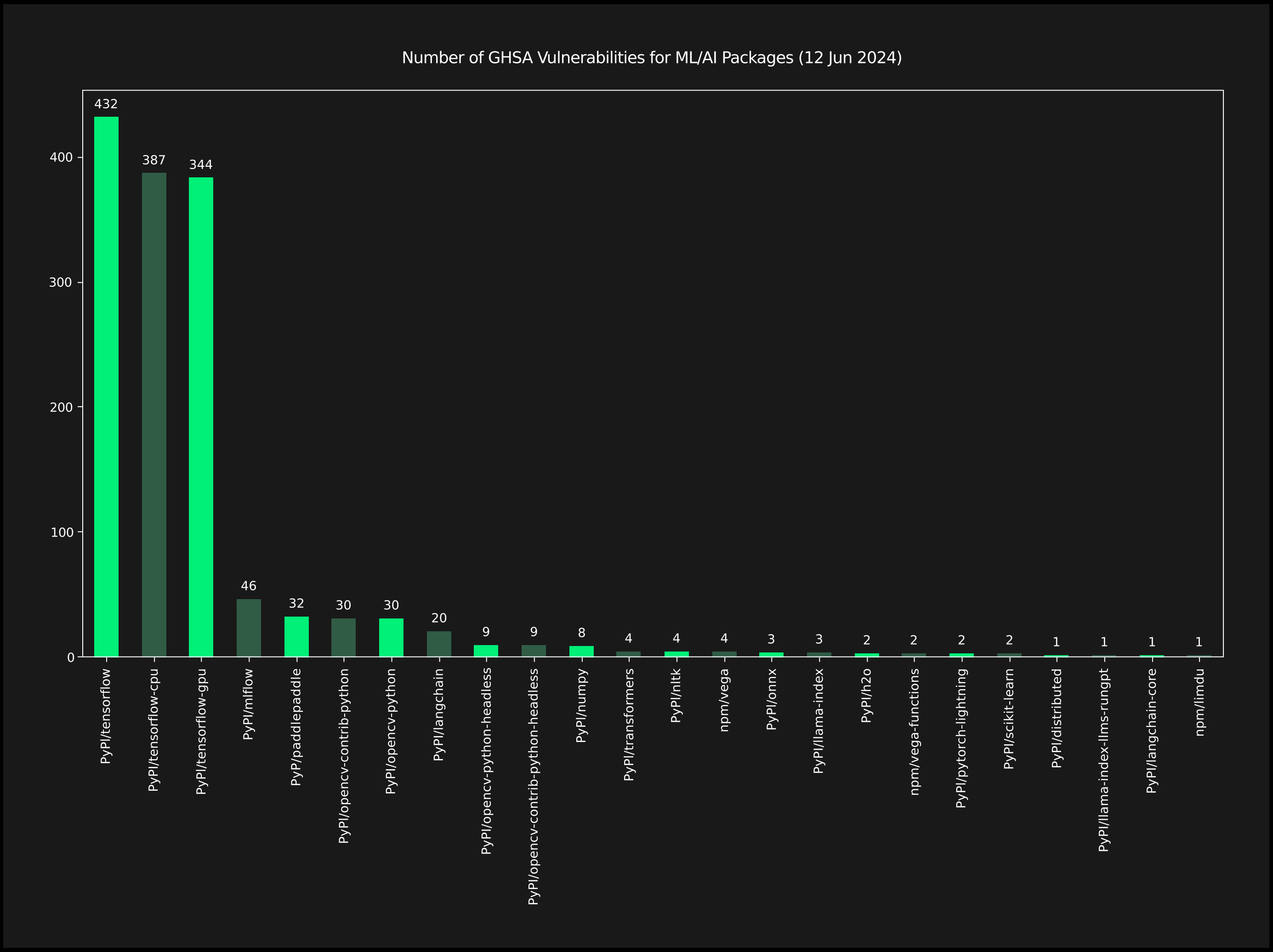
The next two charts illustrate once more the discrepancy between vulnerability databases: They show the number of vulnerabilities for common ML/AI packages – first considering all databases aggregated by OSV, second only considering the advisories from GitHub's Advisory Database (GHAD).

Tensorflow has by far the most vulnerabilities – closely followed by the variants tensorflow-gpu and tensorflow-cpu, which is not surprising since they share most of the code base. The number of vulnerabilities considering all OSV data sources, however, is significantly higher compared to the ones reported only through GHSAs, around 40 advisories.

One example is GHSA-24x6-8c7m-hv3f, which only lists tensorflow as affected. The advisories PYSEC-2021-227, PYSEC-2021-518, PYSEC-2021-716, however, mentioned by OSV as aliases of the GHSA, additionally mark tensorflow-gpu/cpu as affected. In the case of pandas, GitHub does not provide a GHSA for the disputed vulnerability CVE-2020-13091.

Open source vulnerability databases can differ significantly in the number of vulnerabilities reported for given packages – illustrated at the example of tensorflow. Users can hardly understand the root cause of such discrepancies, e.g. whether they are due to reporting inconsistencies or disputes.




Number of GHSA Vulnerabilities for ML/AI Packages (12 Jun 2024)
0
100
200
300
400
432
387
344
46
32
30
30
20
9
9
8
4
4
4
3
3
2
2
2
2
1
1
1
1
PyPI/tensorflow
PyPI/tensorflow-cpu
PyPI/tensorflow-gpu
PyPI/mlflow
PyPI/paddlepaddle
PyPI/opencv-contrib-python
PyPI/opencv-python
PyPI/langchain
PyPI/opencv-python-headless
PyPI/opencv-contrib-python-headless
PyPI/numpy
PyPI/transformers
PyPI/nltk
npm/vega
PyPI/onnx
PyPI/llama-index
PyPI/h2o
npm/vega-functions
PyPI/pytorch-lightning
PyPI/scikit-learn
PyPI/distributed
PyPI/llama-index-llms-rungpt
PyPI/langchain-core
npm/llmdu

Part 3: Why Remediating Known Vulnerabilities Is Hard

TL;DR

- 24% of 1250 updates from vulnerable to non-vulnerable component versions (published by the 15 most-problematic libraries after 2016) require a major version update.

- 76% of 1250 updates can be done by updating the minor or patch version. However, the majority of those also contain breaking changes that can potentially break a client – but often only a handful of functions or types out of tens of thousands.

- Whole program call graphs and the analysis of type hierarchies can clarify whether breaking changes of library updates actually matter in a specific application context.

Suppose a vulnerability has been successfully identified and prioritized for being addressed, there are a number of ways to overcome the problem.

More involved solutions require the development of application-specific safeguards, e.g. to sanitize user input before it enters sensitive sinks in the vulnerable OSS component.

Seemingly the most straight-forward solution is to upgrade to a non-vulnerable version of the dependency. However, what sounds easy in principle – after all, you just need to update the version identifier to a non-vulnerable one, right? – can cause compatibility problems and regressions that break an application during developmen

Piece of Cake: Pulling a Non-vulnerable Version

There are differences in regard to updating vulnerable dependencies depending on the ecosystem, but in the following we will focus on Java only.

For what concerns direct dependencies, the non-vulnerable version identifier can be simply entered in the respective manifest file, e.g., the Maven POM file.

In case of transitive dependencies, it is advisable to check whether there exists a new version of a direct dependency that pulls in a non-vulnerable version of the affected component. However, this can result in updating multiple other components as well, which of course increases the update risk.

If a direct dependency update is not possible or wanted, the update of the transitive dependency can be more involved, depending on the foresight of the OSS project maintainer:

- If they anticipated that their dependencies can become vulnerable at some point in time, they most likely created dependencyManagement sections – in case of Maven – and used properties to specify dependency versions. In this case, updating to a non-vulnerable version is as easy as overriding the respective property, and the Spring Boot project is a good example for this practice.

- If they did not follow this best practice, the application developer can create a direct dependency using the non-vulnerable version identifier, such that the package manager's resolution logic will give priority to the version specified directly.

Tough Nut: Breaking changes

However, what really matters to application developers is whether the new component version is compatible with the previous one, which has two dimensions: **API compatibility**, which concerns the public functions and types exposed by the component and used by its consumers, and **semantic compatibility**, which boils down to showing equivalent functional behavior.

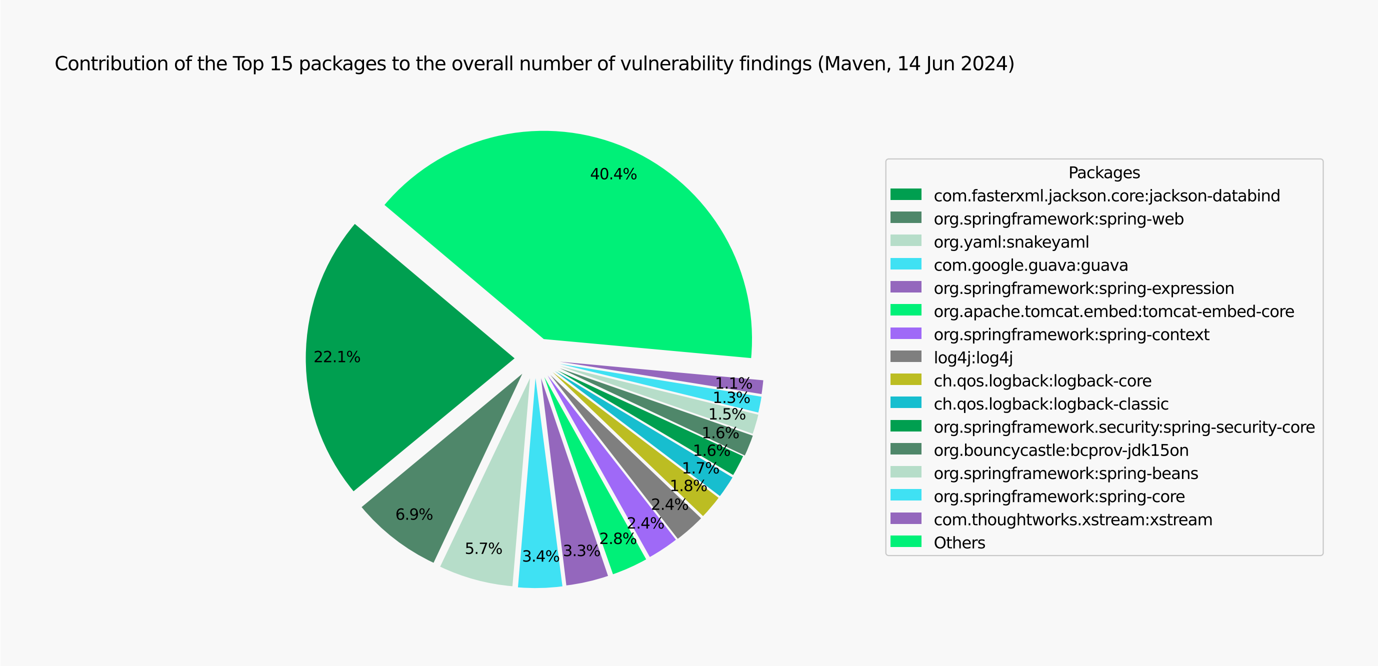
A straightforward example for API incompatibilities is the removal of a function that existed in an older version, but has been deleted in the new, non-vulnerable version of the component. A call of the application into this function will not be possible once the new version is used, and will result in a compile or runtime error.

SemVer is meant to indicate whether there are any incompatibilities between two versions: Two versions differing in the major version element are expected to contain breaking changes, whereas versions that only differ in the other elements, e.g. minor or patch, are supposed to be compatible. However, according to Ochoa et al., "30.5% of non-major releases do not conform to semver" [Ochoa 2018].

Developers' fear of breaking changes are a major inhibitor to upgrade from vulnerable to non-vulnerable component versions [Mirhosseini 2017].

In the following, we look at the problem of breaking changes more closely, at the example of those 15 Java components that bring in the most vulnerabilities for our customers (same pie chart as above).

In the following, we look at the problem of breaking changes more closely, at the example of those 15 Java components that bring in the most vulnerabilities for our customers (same pie chart as above).


Contribution of the Top 15 packages to the overall number of vulnerability findings (Maven, 14 Jun 2024)

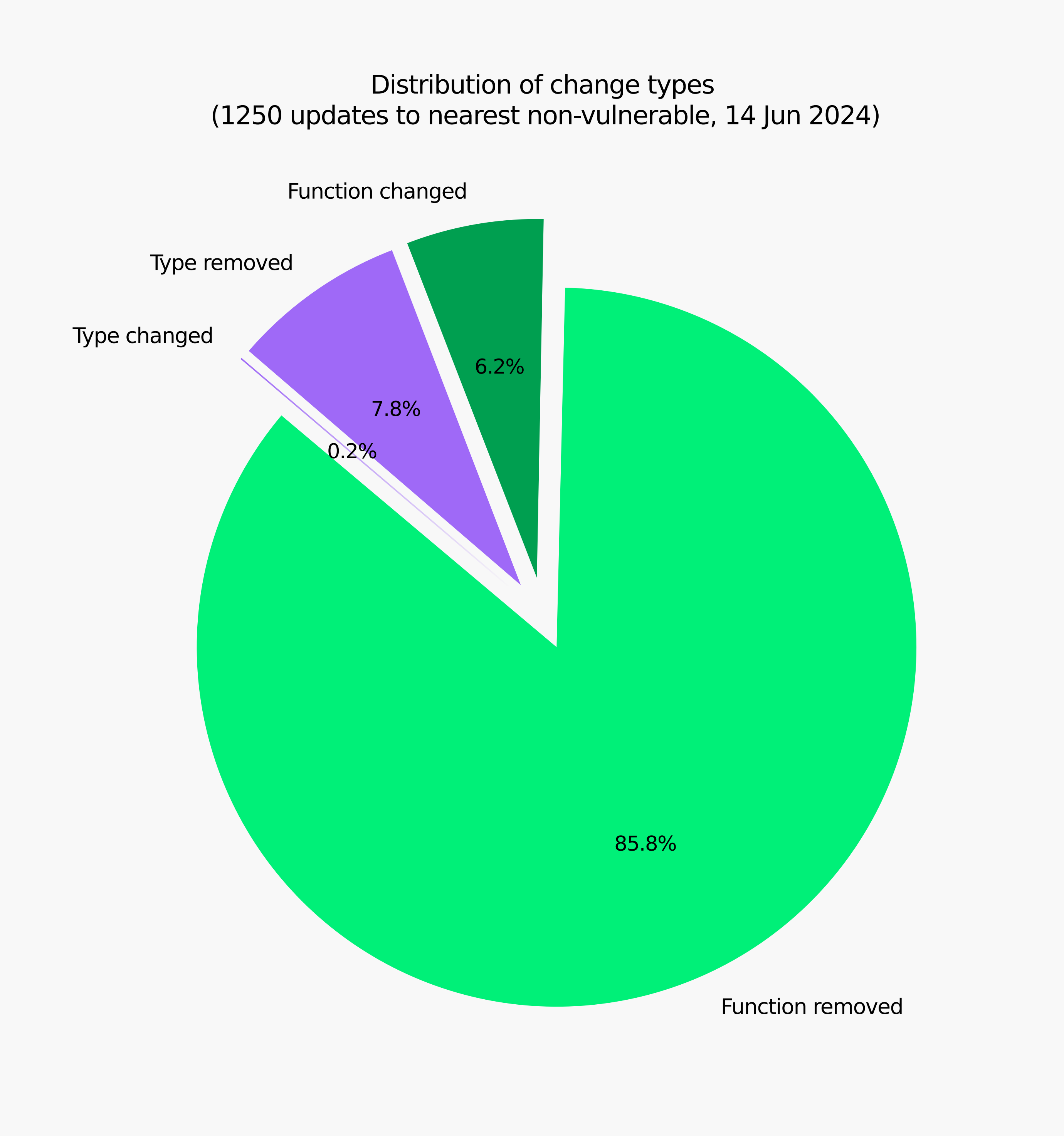
Holy Grail: Change Impact Analysis

To better understand the prevalence of breaking API changes in the versions of those components that matter for our customers, we extended our existing analysis framework to detect the following four code changes between any two component versions: The removal of functions and types such as interfaces or classes, as well as the change of functions' and types' visibility, e.g. from public to private, which determines whether a function or type is visible and accessible from outside the component.

Then, we obtained all the versions of the 15 components depicted in the above pie chart (2649), dropped versions published prior to 2017 (1639 remained) and associated known vulnerabilities, which leaves us with 1265 versions that have known vulnerabilities.

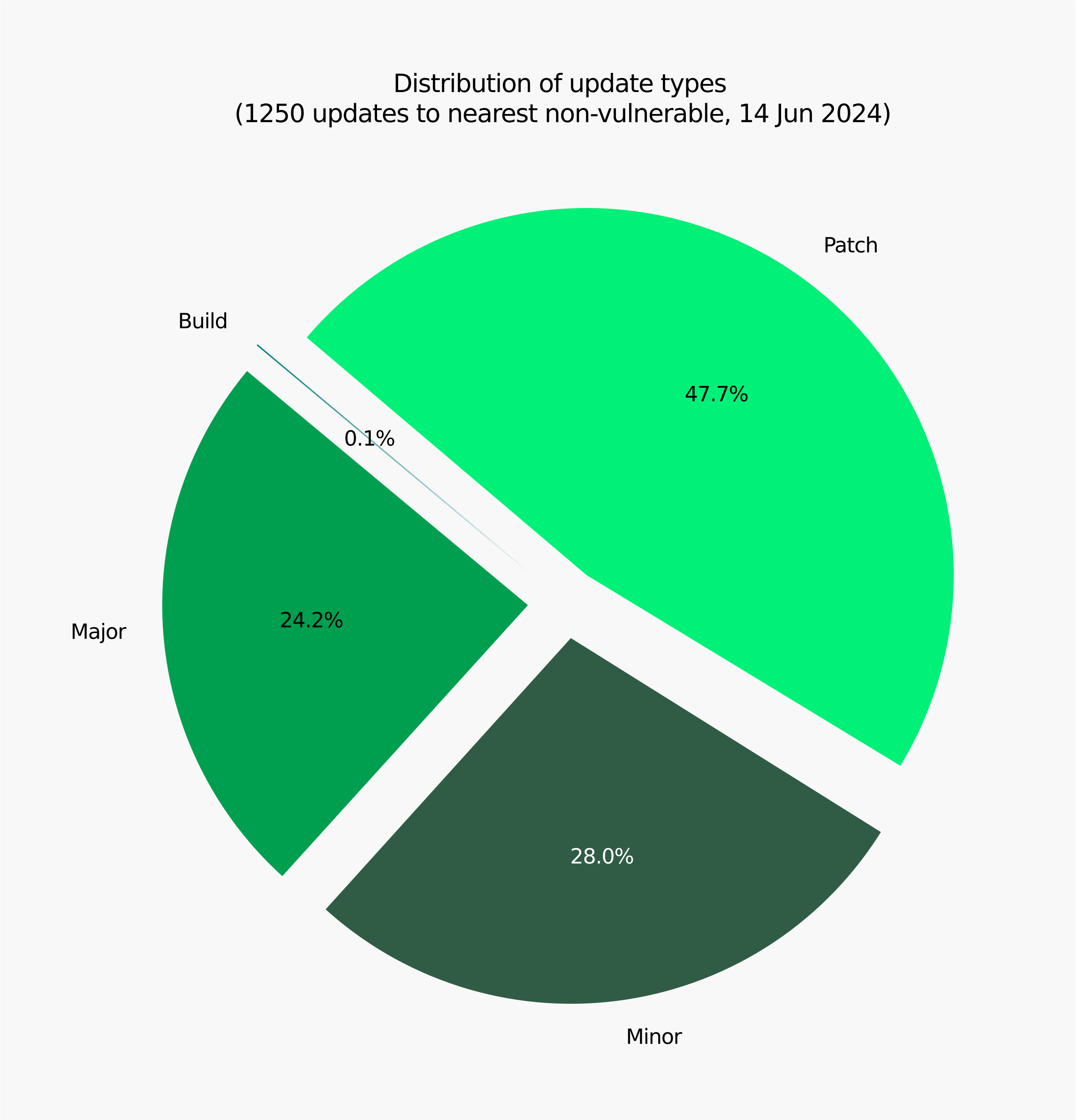
Developers depending on any of those versions may want to update to **the nearest greater version that has no vulnerabilities**. This allows them getting rid of all the component's vulnerabilities, while minimizing code changes between the current and updated version (compared to updating to more recent versions). Nearest non-vulnerable versions are available for **1250** vulnerable component versions (the 15 versions that cannot be updated belong to org.bouncycastle:bcprov-jdk15on, which does not have a vulnerability-free version), and the following charts look at them from different angles...

First, it is interesting to see that functional removals are by far the most prominent change between the respective versions, which aligns well with the observation made by Ochoa et al.

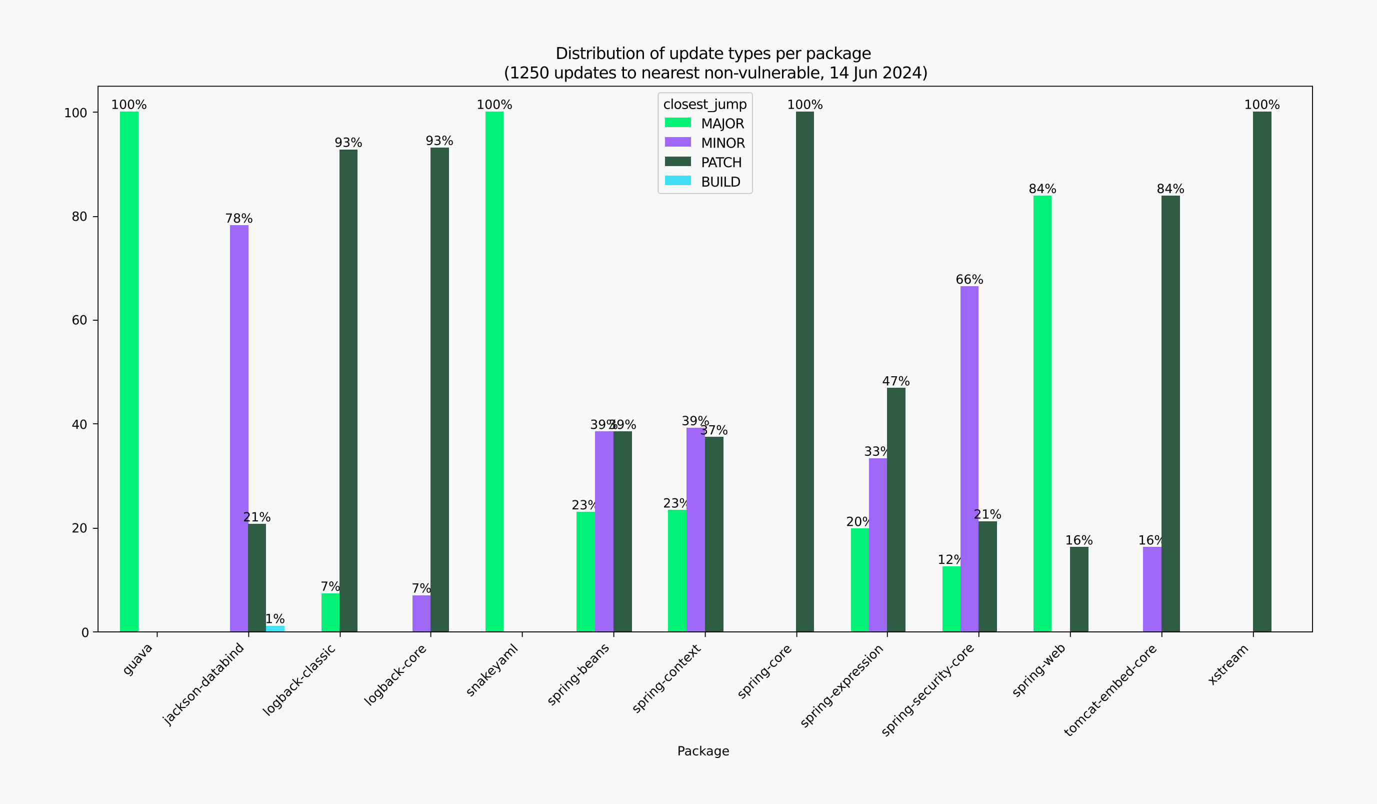


The following chart shows how many of those updates to the nearest non-vulnerable version come with increases in the major, minor, patch or build versions respectively, which is a first indicator whether or not the updates can be expected to break the application.

It turns out that 24% of the updates require to move to another major version, which basically means that the respective old release branch is not maintained any longer. One such example is spring-web@4.3.30, released in Dec 2000, which was the last release of the 4.3.x release branch before it reached end of life. Fixing its vulnerabilities requires updating to the maintained 5.3.x and 6.x release branches.



The following chart looks at this data per package, showing that most of the vulnerable versions (84%) of spring-web released after 2016 require major version updates when wanting to move to vulnerable-free ones. The situation looks better for packages like logback-core and logback-classic, where most of the updates can be done by bumping the patch version, e.g. from logback-core@1.2.9 (released in Feb 2017) to version 1.2.13 from Dec 2023.

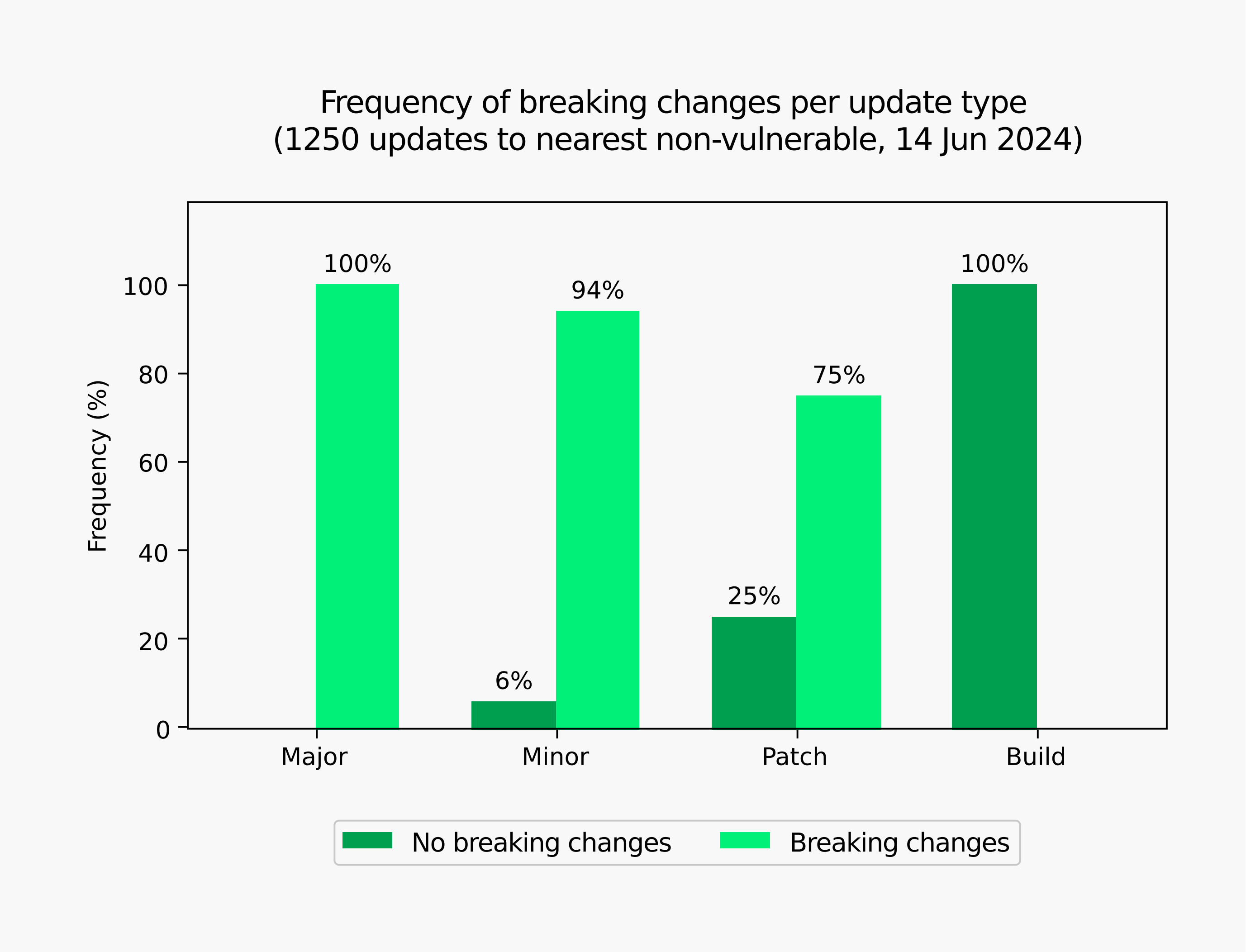


The following chart shows breaking changes per update type: All major version updates contain breaking changes, but none of the build updates, which is inline with the SemVer versioning scheme.

What's more surprising, though inline with the observations of Ochoa et al., is that minor and patch version updates also contain breaking changes. This phenomenon explains why supposedly "safe", non-breaking changes – at least according to SemVer – create problems for applications. In other words: **It is not sufficient to only compare version identifiers to conclude whether an update is seamless or not.**

Instead, program analysis techniques such as call graphs and type hierarchies can help to understand the actual impact of component updates on both first and third party code – no matter whether the updated component is a direct or transitive dependency.

The following chart looks at this data per package, showing that most of the vulnerable versions (84%) of spring-web released after 2016 require major version updates when wanting to move to vulnerable-free ones. The situation looks better for packages like logback-core and logback-classic, where most of the updates can be done by bumping the patch version, e.g. from logback-core@1.2.9 (released in Feb 2017) to version 1.2.13 from Dec 2023.



4

Part 4: Software Composition Analysis and its role in dependency management

SCA Tools Must Do More to Support Remediation Efforts

A good SCA tool surfaces all potential risks to an application, regardless of whether those risks can be exploited based on the organization's unique practices. For most organizations, fixing every single CVE or other advisory that potentially affects their applications is just not feasible — or at least not a good use of limited development and security resources. That means it's essential to determine which vulnerabilities have the potential to cause serious harm to the environment.

Historically SCA tools enable filtering based on CVSS score, with the common practice being to prioritize the criticals and highs for remediation. But this approach is problematic because a CVSS score measures vulnerability severity, not risk. A more accurate way to measure risk to the organization is based on exploitability and probability of exploit. In this section, we looked at several context-based methods to determine which has the best "noise reduction."

Through this research, we learned:

- Function-level reachability is the most important prioritization capability.
- EPSS is the second-most important prioritization technique.
- Prioritizing risks is only the first step; organizations need help prioritizing remediation efforts.

Function-level reachability is the most important prioritization capability.

> Less than 9.5% of vulnerabilities are exploitable at the function level.

For a vulnerability in an open source library to be exploitable, there must at minimum be a call path from the application you write to the vulnerable function in that library. By examining a sample of our customer data where reachability analysis is being performed, we found this to be true in fewer than 9.5% of all vulnerabilities in the seven languages we support this level of analysis for at the time of publication (Java, Python, Rust, Go, C#, .NET, Kotlin, and Scala). This method alone - called reachability analysis - offers a significant reduction in remediation costs because it lowers the number of remediation activities by an average of 90.5% (with a range of approximately 76–94%), making it by far the most valuable single noise-reduction strategy available.

While function-level reachability can be determined manually, it is a resource-intensive activity that AppSec and engineering teams are not staffed to support at scale. Without an SCA tool that reliably and accurately performs this analysis, teams typically prioritize solely by CVSS and often end up performing upgrades even if the vulnerabilities aren't exploitable. The data underscores the importance for SCA tools to support this prioritization method.

Other context-based strategies we researched (excluding vulnerabilities that are only relevant for non-production code and those without a fix available) can reduce scope further where they are relevant. However; no combination of these strategies has as significant an impact on scope as function-level reachability. If you have an SCA tool (like Endor Labs) that can apply multiple strategies without significant trade-offs, they are still worthwhile to implement in addition to function-level reachability.

EPSS is the second-most important prioritization technique.

> Programs that combine reachability and EPSS see a noise reduction of 98%.

Exploit Prediction Scoring System (EPSS) is a data-driven method to estimate the likelihood (probability) that a software vulnerability will be exploited in the wild. After function-level reachability, EPSS is the next most impactful prioritization method. We found that 4 out of 5 reachable vulnerabilities have a 1% or less predicted chance of being exploited. Programs that combine reachability and EPSS are therefore able to prioritize their limited resources toward repairing an average of just 2% of total reported vulnerabilities.

While EPSS is a statistical model, and therefore can't be used to guarantee anything about exploitability of a vulnerability, it is a useful tool for making predictions about where to focus remediation activities. Essentially, focusing on items with higher EPSS probability scores helps in two key ways:

- Addressing your highest-likelihood items first reduces total risk faster
- Reducing total risk faster means that if remediation resources are redirected due to other urgent work, such resource reallocations have less overall impact on your organization's risk footprint

When combined with function-level reachability analysis data and other context-based scoping strategies, EPSS prioritization is often so effective that additional, higher-effort prioritization strategies (such as conducting Environmental and Temporal CVSS scoring exercises to determine severity in your environment) are often unneeded. This saves vulnerability analysis costs for your organization.

Prioritizing risks is only the first step; organizations need help prioritizing remediation efforts.

> 95% of version upgrades contain at least one breaking change.

Upgrading dependencies can be as simple as updating a version number in a manifest file; but many times it is much more complex than that due to “breaking changes” — changes in the way the dependency works that require application developers to make changes to their own code. The research, implementation, and testing of the upgrade and the required code changes to support it are the largest factor in cost to remediate a particular issue.

Major version increments (e.g. 1.x to 2.x) are very often indicators of significant breaking changes. In some cases, major version upgrades to dependencies can be multi-month efforts for dev teams. For security teams, this means a two key things:

- Being prepared to weigh the risk of a particular vulnerability against the risk of performing the upgrade, then providing mitigation strategies and/or granting security exceptions where appropriate
- Planning for and communicating remediation timelines, especially where making the major upgrades may cause SLAs or regulatory clocks to be exceeded

We found that remediation of vulnerabilities in dependencies requires applying the appropriate fix, which requires a major version update 24% of the time. Major version updates are significantly more likely to require significant changes to customer applications, increasing the cost for software engineering teams to remediate security issues. Even with aggressive prioritization, there are opportunities to lower remediation costs further.

Further, of the remaining updates that can be fixed with a minor update or patch: 94% of the time, minor version updates contain at least one change that could potentially break a client , and patch version updates have this risk 75% of the time.

SCA Tools Need Inputs from Public and Private Vulnerability Databases

One of the most essential roles of an SCA tool is to correlate a list of vulnerabilities with your application code to see if you’re potentially exposed. This requires the tool to generate an accurate software inventory (including both direct and transitive dependencies) and have access to a robust vulnerability database. Because public vulnerability databases are a de facto information source for SCA tools, we looked at the reliability of those databases as primary data sources.

Through this research, we learned:

- Tracking leading indicators of risk can offset vulnerability disclosure gaps
- Vendor enrichment must supplement public advisory databases

Tracking leading indicators of risk can offset vulnerability disclosure gaps

> 69% of vulnerability advisories are published after a fix has been released, with a median delay of 25 days between public patch availability and advisory publication (Imitaz, Khanom, Williams).

When open source projects fix a vulnerability, it happens in the open, which has a side-effect of slightly raising the risk of exploitation by "leaking" the existence (and to some extent, the nature) of the vulnerability. However, it's generally impractical for organizations to monitor every open source package they use to determine whether any changes might be related to security risk, so defenders rely on public advisories.

Organizations have some control over how quickly they respond to advisories, but there is a period of elevated risk between release of the fix and publication of the advisory— and this is outside of what defenders can control. When this gap is measured in weeks or more, as is often the case, it poses a unique challenge for defenders.

While addressing this gap is not the highest-priority issue for every organization, those with mature dependency risk management programs can begin to address it by monitoring projects for behaviors and attributes that tend to predict risks. For example, the OWASP Open Source Top 10 list contains several leading indicators of risk that can be detected by an SCA tool. This allows defenders to drive adoption of lower-risk dependencies early in the SDLC, as well as target resources toward more-intensive proactive monitoring of the riskiest projects.

Vendor enrichment must supplement public advisory databases

> Only 2% of public advisories contain information about which library functions contain the vulnerability

In order to effectively triage vulnerabilities in dependencies, and to recommend mitigation strategies and prioritize remediation activities, AppSec and related teams need complete and accurate data about what components are affected, and where specifically the vulnerability resides in the affected component. Code-level vulnerability assessment requires knowing the affected functions, either explicitly or through inference from the code changes that overcome the vulnerability ("fix commits").

Unfortunately, it's fairly common for advisory databases to have incomplete or incorrect information:

- **Advisories can incorrectly identify which component contains the vulnerability.** For example, listing an open-source application as impacted but not the library which contains the vulnerability (resulting in false negatives), or identifying the wrong component entirely (resulting in both false negatives and false positives)
- **Advisories can have incorrect or incomplete data about which versions are affected.** Affected versions may not be listed because the reporter never tested them (false negatives), and safe versions might be listed as affected (false positives)
- **It is rare for advisories to have sufficient detail to identify the vulnerable function(s).** This makes it challenging for defenders to devise mitigation strategies, verify relevance and impact, and provide accurate prioritization inputs

In our research, we found:

- Only 2% of public advisories contain information about which library functions contain the vulnerability. Slightly more than half (51%) of advisories provide information about fix commits.
- Public advisory databases have incomplete and inaccurate data about affected components. 28% of advisories have either incorrect or incomplete data that can lead to false positives and false negatives for risks, as well as wasting time remediating incorrectly.

Dependency risk management programs typically rely on open source or commercial tools to enable this analysis, but since many tools do not enrich public advisory data (i.e. about affected components, affected versions, and vulnerable functions), defenders can not assume tools provide better quality data than a given advisory database. This gap is why Endor Labs continues to invest in high-quality enrichment of vulnerability data, including analysis and correction of affected component and version data and reliable identification of vulnerable functions.

AI Projects Pose Unique Dependency Management Challenges

"But what about AI?" has become a recurring joke at industry events, but the reality is that AI and machine learning (ML) are changing the way developers code their applications. For this topic, we sought to understand how the use of AI/ML impacts SCA findings and whether vulnerability reporting is adequate for AI libraries.

Through this research, we learned:

- Awareness of phantom dependencies — which are more common in AI and ML software projects — is critical
- Vulnerability reporting is less consistent for AI libraries

Awareness of phantom dependencies — which are more common in AI and ML software projects — is critical

> 27% of organizations have a significant phantom dependency footprint, and within that group, over 56% of reported library vulnerabilities are in their phantom dependencies.

Phantom dependencies are dependencies that are installed outside the normal application build and dependency management process, and are often invisible to security tools as they don't appear in manifests and lock files. This happens because many dependency risk management programs use the software project's dependency manifests (files like pom.xml or package.json) to identify the direct dependencies of that project. This strategy misses libraries and frameworks that projects rely on existing in a target deployment environment (for example, relying on a container or host having particular libraries or tools already installed).

We surveyed our customers' development environments and found such "phantom dependencies" are not a significant part of the dependency footprint for most of our customers. However, the 27% of our customer environments that do have a significant number of phantom dependencies find that those dependencies tend to be a disproportionate source of vulnerabilities, likely because previous risk management programs have not had adequate visibility to phantom dependencies. We note that projects with a significant AI or ML presence are significantly more likely to make use of phantom dependencies, and we surmise that common practices and considerations with AI development lend themselves to this pattern more than traditional projects.

Vulnerability reporting is less consistent for AI libraries

> Reported vulnerability counts for a significant AI and ML library can vary by as much as 10% between public advisory databases.

We compared various advisory databases that have advisories for widely-used AI and ML dependencies and found that there is a very high chance that a given advisory database is missing up to 10% of the advisories related to a given library.

This variance highlights the importance of correlating multiple advisory sources to ensure correct coverage; this is a good practice regardless of whether an organization has invested in AI features, but essential for those that have.

Footnotes and References

Organized in order of appearance

- S. Jha, “Anomaly Detection Based on Weakly Supervised Learning for Industrial Applications,” IEEE Access, vol. 10, pp. 55343-55355, 2022. Available: https://ieeexplore.ieee.org/document/9792380.

- Endor Labs, “5 Types of Reachability Analysis and Which Is Right for You,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/5-types-of-reachability-analysis-and-which-is-right-for-you. [Accessed: 22-Aug-2024].

- Endor Labs, “What is Reachability-Based Dependency Analysis,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/what-is-reachability-based-dependency-analysis. [Accessed: 22-Aug-2024].

- NIST, “NVD Program Transition Announcement,” NVD, 05-May-2024. Available: https://web.archive.org/web/20240505091722/https://nvd.nist.gov/general/news/nvd-program-transition-announcement. [Accessed: 22-Aug-2024].

- Endor Labs, “Dependency Resolution in Python: Beware the Phantom Dependency,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/dependency-resolution-in-python-beware-the-phantom-dependency. [Accessed: 22-Aug-2024].

- Intel, “neural_compressor/contrib/strategy/sigopt.py,” GitHub, [Online]. Available: https://github.com/intel/neural-compressor/blob/master/neural_compressor/contrib/strategy/sigopt.py#L125. [Accessed: 22-Aug-2024].

- OSV, “Open Source Vulnerability Database,” OSV.dev, [Online]. Available: https://osv.dev/list?ecosystem=PyPI&q=numpy. [Accessed: 22-Aug-2024].

- S. M. Larson, “Security Developer in Residence Weekly Report #16,” Seth M. Larson’s Blog, [Online]. Available: https://sethmlarson.dev/security-developer-in-residence-weekly-report-16. [Accessed: 22-Aug-2024].

- PyCode, “PyCode Statistics,” PyCode, [Online]. Available: https://py-code.org/stats. [Accessed: 22-Aug-2024].

- E. Bodden, D. Plattner, and H. Hojjat, “Identifying Vulnerabilities in Third-Party Dependencies,” Software Engineering Research Group, 2021. Available: https://www.bodden.de/pubs/dph+21identifying.pdf.

- NIST, “NVD Program Transition Announcement,” NVD, [Online]. Available: https://nvd.nist.gov/general/news/nvd-program-transition-announcement. [Accessed: 22-Aug-2024].

- Google, “Covered Ecosystems,” OSV, [Online]. Available: https://google.github.io/osv.dev/data/#covered-ecosystems. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-r95h-9x8f-r3f7,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-r95h-9x8f-r3f7. [Accessed: 22-Aug-2024].

- Maven, “Artifact com.vaadin/vaadin-bom/24.4.1,” Search.maven.org, [Online]. Available: https://search.maven.org/artifact/com.vaadin/vaadin-bom/24.4.1/pom. [Accessed: 22-Aug-2024].

- Apache Maven, “Introduction to Dependency Mechanism,” Maven, [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#bill-of-materials-bom-poms. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-hw7r-qrhp-5pff,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-hw7r-qrhp-5pff. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-56pw-mpj4-fxww,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-56pw-mpj4-fxww. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability PYSEC-2023-175,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/PYSEC-2023-175. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-jgpv-4h4c-xhw3,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-jgpv-4h4c-xhw3. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability OSV-2022-1074,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/OSV-2022-1074. [Accessed: 22-Aug-2024].

- Unit 42, “The State of Exploit Development,” Palo Alto Networks, [Online]. Available: https://unit42.paloaltonetworks.com/state-of-exploit-development/. [Accessed: 22-Aug-2024].

- S. Hendrikse, “Understanding Deep Learning,” arXiv.org, [Online]. Available: https://arxiv.org/pdf/2006.15074. [Accessed: 22-Aug-2024].

- A. Saxe et al., “Learning Deep Linear Networks,” arXiv.org, [Online]. Available: https://arxiv.org/pdf/2112.06804. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-hm9v-vj3r-r55m,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-hm9v-vj3r-r55m. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability CVE-2023-36807,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/CVE-2023-36807. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability GHSA-35jh-r3h4-6jhm,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-35jh-r3h4-6jhm. [Accessed: 22-Aug-2024].

- OSV, “Vulnerability CVE-2021-23337,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/CVE-2021-23337. [Accessed: 22-Aug-2024].

- IEEE, “Deep Learning for Industrial Applications,” IEEE Xplore, [Online]. Available: https://ieeexplore.ieee.org/iel7/8013/5210089/10381645.pdf. [Accessed: 22-Aug-2024].

- Endor Labs, “Why SCA Tools Can’t Agree If Something Is a CVE,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/why-sca-tools-cant-agree-if-something-is-a-cve. [Accessed: 22-Aug-2024].
- NIST, “CVE-2023-41080,” NVD, [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-41080. [Accessed: 22-Aug-2024].
- Apache, “Thread Discussion on Apache List,” Apache Mailing List, [Online]. Available: https://lists.apache.org/thread/71wvwprtx2j2m54fovq9zr7gbm2wow2f. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability GHSA-7fmw-85qm-h22p,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-7fmw-85qm-h22p. [Accessed: 22-Aug-2024].
- Microsoft, “CVE-2023-36566,” MSRC, [Online]. Available: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2023-36566. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability GHSA-vm2m-7hpw-fpmq,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-vm2m-7hpw-fpmq. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability GHSA-pj27-2xvp-4qxg,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-pj27-2xvp-4qxg. [Accessed: 22-Aug-2024].
- B. Dong, “Exploiting Vulnerabilities in Open Source Software,” USENIX, [Online]. Available: https://www.usenix.org/system/files/sec19-dong.pdf. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability GHSA-m7jv-hq7h-mq7c,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/GHSA-m7jv-hq7h-mq7c. [Accessed: 22-Aug-2024].
- GitHub, “Advisory GHSA-24x6-8c7m-hv3f,” GitHub, [Online]. Available: https://github.com/advisories/GHSA-24x6-8c7m-hv3f. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability PYSEC-2021-227,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/PYSEC-2021-227. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability PYSEC-2021-518,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/PYSEC-2021-518. [Accessed: 22-Aug-2024].
- OSV, “Vulnerability PYSEC-2021-716,” OSV.dev, [Online]. Available: https://osv.dev/vulnerability/PYSEC-2021-716. [Accessed: 22-Aug-2024].
- Spring, “Appendix: Dependency Versions,” Spring Documentation, [Online]. Available: https://docs.spring.io/spring-boot/appendix/dependency-versions/properties.html#appendix.dependency-versions.properties. [Accessed: 22-Aug-2024].
- SemVer, “Semantic Versioning Specification,” SemVer.org, [Online]. Available: https://semver.org/. [Accessed: 22-Aug-2024].
- A. K. Ghosh et al., “Adversarial Attacks on Neural Networks,” arXiv.org, [Online]. Available: https://arxiv.org/pdf/2110.07889. [Accessed: 22-Aug-2024].

- NSF, “Research in Software Vulnerability Mitigation,” NSF Public Access Repository, [Online]. Available: https://par.nsf.gov/servlets/purl/10057926. [Accessed: 22-Aug-2024].

- Spring Projects, “Spring Framework Versions,” GitHub, [Online]. Available: https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions. [Accessed: 22-Aug-2024].

- Endor Labs, “5 Types of Reachability Analysis and Which Is Right for You,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/5-types-of-reachability-analysis-and-which-is-right-for-you. [Accessed: 22-Aug-2024].

- Endor Labs, “EPSS Exploit Prediction & Reachability Analysis,” Endor Labs, [Online]. Available: https://www.endorlabs.com/learn/epss-exploit-prediction-reachability-analysis. [Accessed: 22-Aug-2024].

- OWASP, “OWASP Top Ten for Open Source Software,” OWASP, [Online]. Available: https://owasp.org/www-project-open-source-software-top-10/. [Accessed: 22-Aug-2024].