ENDOR LABS

STATION 9 BY ENDOR LABS

State of Dependency Management 2023

Henrik Plate

July 2023

Introduction

Will LLMs and AI eat the AppSec world?

As modern software trends toward distributed architectures, microservices, and extensive use of third party and open source components, dependency management only gets harder. Our latest report explores emerging trends that software organizations need to consider as part of their security strategy.

There has been an explosion of interest in large language model (LLM) based AI platforms. Hundreds of packages designed to integrate these capabilities into other applications have emerged in just a few months since ChatGPT's API was released. Organizations need to be careful when selecting packages due to their lack of historical data and popularity which attracts attackers. Focusing specifically on LLM applications in the security realm, we found that LLM can effectively create and hide malware and may become a nemesis to defensive LLM applications.

It appears inevitable that organizations will need to document the components and vulnerabilities their applications include, using something like an SBOMs and accompanying VEX documents. But applications often use only a small portion of the open source components they integrate, and developers rarely understand the cascading dependencies of components. In order to satisfy transparency requirements while protecting brand reputation, organizations need to go beyond basic SBOMs. Understanding not only which components are included but which functions are used and which vulnerabilities are exploitable will satisfy transparency requirements, enable a better understanding of risk, improve productivity and drive other cost savings.

Highlights

- Just 5 months after release, ChatGPT's API is used in 900 npm and PyPi packages across diverse problem domains. **70% of those are brand new packages.**

- **Current LLM technologies have a low precision rate of less than 10% when analyzing malware.** While useful in manual workflows, they may never be able to work reliably in autonomous workflows because of legitimate uses of suspicious data flows and techniques, or the adversarial context – with attackers actively trying to evade detection, also with help of LLM recommendations.

- 55% of applications have calls to security sensitive APIs in their code base, **but that rises to 95% when dependencies are included.** Organizations significantly underestimate risk by failing to analyze their use of such APIs through their open source dependencies.

- 71% of typical Java application code is from open source components, **yet apps use only 12% of imported code.** Vulnerabilities in unused code are rarely exploitable; organizations can eliminate or de-prioritize up to 60% of remediation work with reliable insight into which code is reachable throughout an application.

AI and Large Language Models

Explosion of Models and Tools

Takeaways:

- LLMs are hot! Just 5 months after release, ChatGPT's API is used in 900 npm and PyPi packages across diverse problem domains. 70% of those are brand new packages.

AI in general and large language models (LLMs) in particular became a dominating technology topic since the release of ChatGPT on 30 November 2023. Countless startups, collaborations and services were announced following this event, and the general public started recognizing the huge potential of those technologies - both in the positive and negative sense.

As with many other technologies in the past, open source promises to become a serious competitor to commercial service providers. Viable open source solutions exist for a significant share of the AI technology stack [1, 2], from data sets and foundation models to programming frameworks and vector databases.

Python and Javascript represent some of the most popular AI development ecosystems. As part of our continuous efforts to catch malicious package uploads to npm and PyPI, we started tracking calls of newly published packages to the OpenAI API. We of course expected packages to use those APIs, but were surprised when seeing the numbers shown in Figure 1. Following the release of ChatGPT's programming API at the end of January 2023, more than 636 new PyPI and npm packages using this API have been created (203 in June 2023 alone). On top of those come another 276 that existed before and added ChatGPT support.

FIG. 1

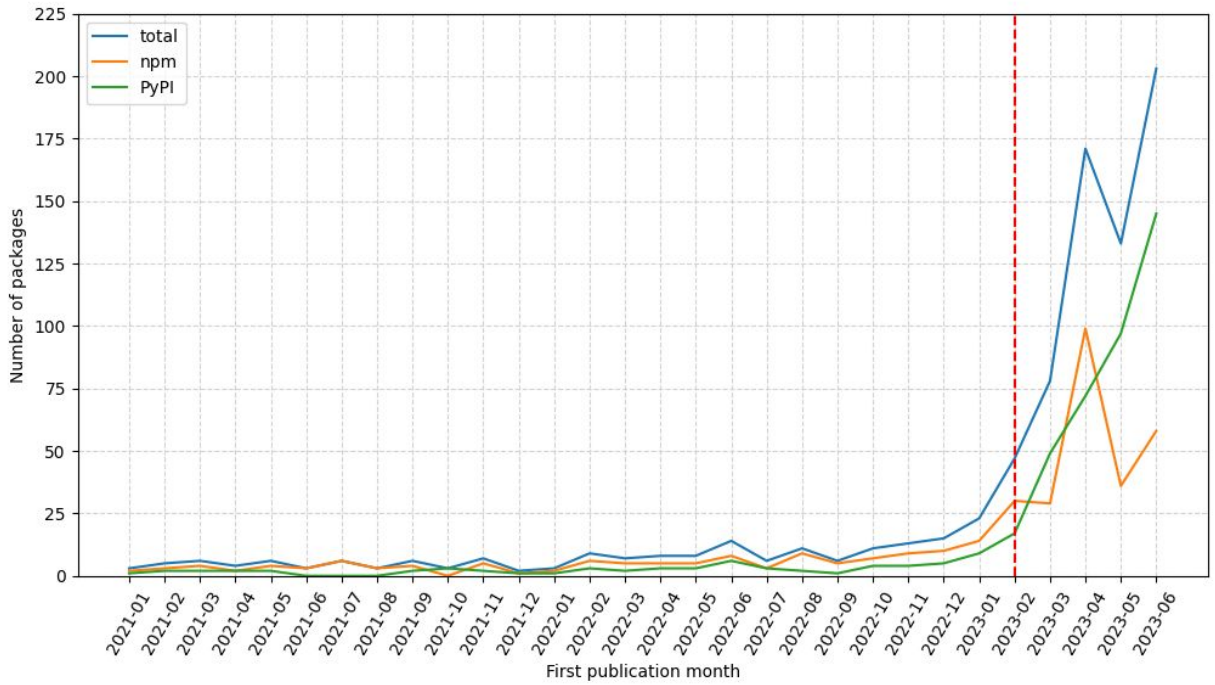

Fig 1 - Number of npm and PyPI projects created between January 2021 and June 2023 that use the OpenAI API

Of course, this is only a small subset of open source projects that made it on npm or PyPI; the number of private projects that experiment with LLMs is certainly much larger.

This observation led us to investigate the security properties of open source AI projects. Spoiler alert: They are as secure or insecure as any other open source project. Before adding one as a dependency, be sure to understand and manage the risk that comes with its use.

Security of Open Source AI Repositories

Takeaways:

- The GitHub repositories of Top-100 open source AI projects reference on average 208 direct and transitive dependencies; 11% rely on 500+ dependencies.
- 15% of these repositories contain 10 or more known vulnerabilities.
- The package distributed by *huggingface/transformers*, for example, has over 200 dependencies (many optional), which include 4 known vulnerabilities.

Open source plays an important role across the AI technology stack, from data sets and foundation models like LLaMA and Vicuna to vector databases and chat bots such as HuggingChat [1, 2, 3]. Still more projects provide tools and developer frameworks needed to integrate AI technologies into software development projects, e.g. LangChain or LlamaIndex, and to improve integration and collaboration among service providers.

To shed some light on any traditional supply chain risks assumed when using such open source AI projects, we scanned the most prominent Top-100 AI projects[1] on GitHub. Because there is no standard measure of prominence, we used the number of times users have "starred" a repository as a measure of popularity or prominence. Tensorflow is the project with the most stargazers (175K+), followed by Auto-GPT (140K+), a project that was created as recently as March 2023.

Our approach to identifying these risks consisted of downloading their source code (using the default branch of their repository), analyzing their manifest files to identify all dependencies, and scanning for vulnerabilities.

[1] Note that there is a lot of variety in these projects: some contain data sets and related Python scripts for data preprocessing, others are libraries to be used in downstream development projects, while others comprise full-blown applications.

Figure 2 illustrates that 77% of the project repositories contain manifest files[2] that reference more than 50 dependencies (directly or transitively). Notable projects with more than 500 dependencies include *huggingface/transformers* (778 dependencies across 51 manifest files, including tests), *spacy* (1044/2) and *pytorch* (1137/45). Having many dependencies is not itself a problem, but risk and complexity generally increase with the number of dependencies.

Figure 3 illustrates that 52% of those project repositories reference known vulnerable dependencies in their manifest files, some up to 100 and more. Whether or not those vulnerabilities pose any real security risk depends on multiple factors.

First and most importantly, we consider whether the manifest file with vulnerable dependencies belongs to a distributed software artifact. Typical examples are packages available on PyPI and downloaded in the context of development projects, or stand-alone applications installed and operated directly by downstream users. But code snippets and template projects may also be problematic.

The *huggingface/transformers* project provides a good example: the GitHub repository references 773 distinct dependencies containing 93 known vulnerabilities. The Python package *transformers*, distributed on PyPI, references 297 dependencies (including all extras) with only 4 known vulnerabilities in its version *4.28.1*.

Secondly, as described in the next section on call graph analysis, vulnerable code may be imported but if it cannot be executed then it represents a much lower risk of exploitation and may be deprioritized in order to reduce review noise and improve remediation productivity.

[2] Manifest files are used by developers to declare direct dependencies, e.g. pom.xml files for Maven or package.json in case of npm. They are processed by package managers to identify and incorporate transitive dependencies to the local development or build environment. Some manifest files correspond to packages uploaded to PyPI or npm, while others are for demos or tests.

Fig. 2

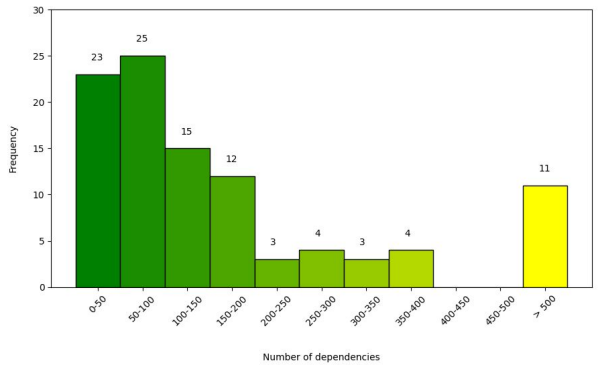

Fig 2 - Top-100 AI repositories: Number of dependencies

Fig. 3

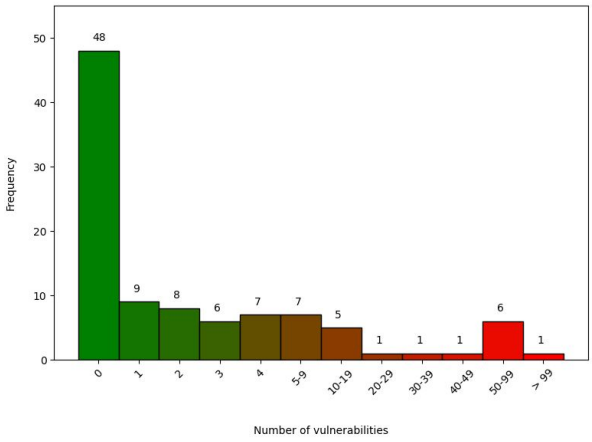

Fig 3 - Top-100 AI repositories: Number of known vulnerabilities

Overall, these numbers suggest that AI projects that produce software artifacts pose the same security challenges for consumers as any other open source project: they have loads of dependencies, many with known vulnerabilities, but the actual risk posed by these packages and vulnerabilities depends on how a given package is used by the consumer.

LLM-assisted malware assessments

Takeaways:

- LLMs are difficult to adapt to malware review and security-related use cases, and can be a double-edged sword.
- The LLMs of two major AI players, gpt-3.5-turbo from OpenAI and text-bison from Google Vertex AI, show 89% agreement in their assessments of potentially malicious code snippets.
- OpenAI GPT3.5 and Vertex AI text-bison do a poor job scoring the malware potential of suspicious code snippets on a scale from 0-9 (3.4% and 7.9% precision, respectively).

Malicious packages continue to be published on package repositories like npm and PyPI, up to the point that PyPI temporarily suspended the registration of new users and packages in May 2023 [4]. Unsurprisingly, both the compromise of legitimate packages and name confusion attacks rank in our Top-10 open source dependency risks [5]. Attackers use techniques like typo-squatting to lure developers into installing malware, or infect legitimate packages using credential stuffing.

Endor Labs continuously scans new packages published on npm and PyPI to identify such malware and alert the respective package repository and prevent further distribution. Our approach is comparable to what's done in academic literature [6, 7, 8]: we gather and analyze information about each package to identify risk markers. Some focus on general information, such as activity patterns and history, and others consider technical artifacts such as syntax trees and data flows. To evaluate how LLMs can be used to help classify malware, we started with the models from OpenAI and later added Google Vertex AI. For consistency, both LLMs were presented with identical prompts[3] to rate the malicious potential of a suspicious code snippet on a scale from 0 (benign) to 9 (malicious).

We were interested to learn how consistent their results were. Figure 4 shows the risk assessment differences for 3374 test cases presented to both LLMs. We consider a scoring difference of 0 or 1 to be agreement, meaning that they agree in 89% of the cases. In 103 cases (3%), however, the rating differs by 5 or more points.

[3] The prompt essentially provides some examples for malicious behavior, asks for the potential impact on victims and includes a pre-processed version of the suspicious snippet (in which comments have been removed to minimize the likelihood of prompt injection attacks).

Fig. 4

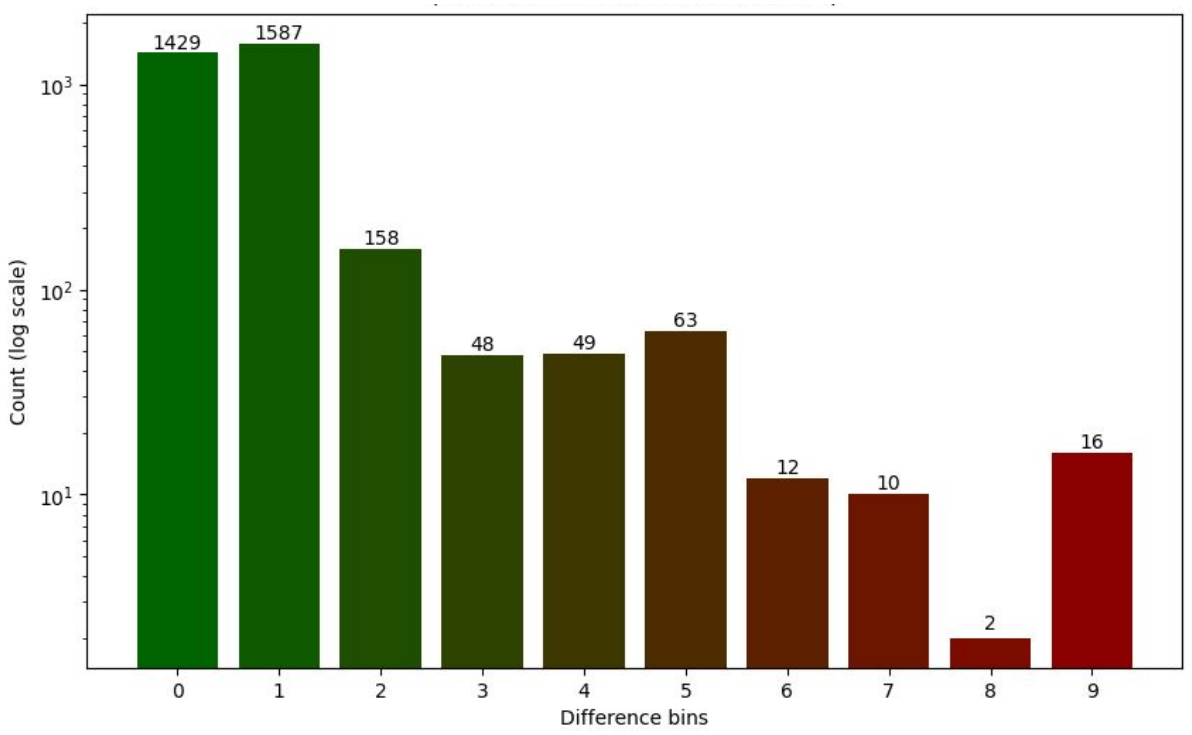

Fig 4 - Risk score difference between OpenAI (gpt-3.5-turbo) and Vertex AI (text-bison): 3374 assessments on a scale 0-9

We also wanted understand how precisely they rated maliciousness. Given that software teams are generally overwhelmed with security findings to investigate, we focused on results that would likely add to that burden. That is, we were more interested in false positives than false negatives.

Figures 5 and 6 show the confusion matrices[4] for the respective models for all cases where (a) the rating differs by 5 or more points or (b) they both classified a snippet as malicious, i.e. they both gave a risk score of 5 or higher.

[4] A confusion matrix is a specific table layout used in machine learning for understanding the performance of a classification model. It visualizes the types of errors made by the model, including false positives, false negatives, true positives, and true negatives, allowing for the in-depth analysis of the model's accuracy, precision, and recall.

Fig. 5

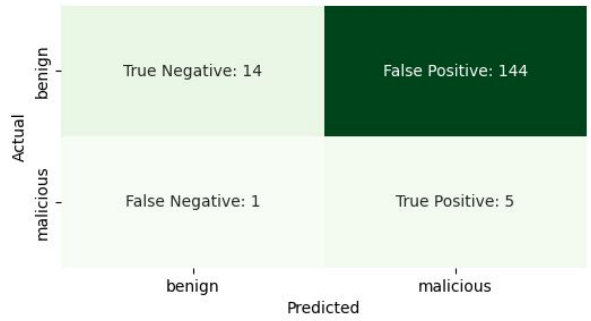

Fig 5 - Confusion matrix for OpenAI gpt-3.5-turbo (precision = 3.4%)

Fig. 6

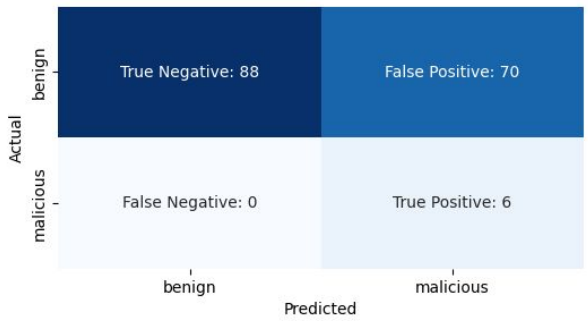

Fig 6 - Confusion matrix for Vertex AI text-bison (precision = 7.9%)

Gpt-3.5-turbo has a lot of false-positives, which results in a low precision of just 3.4%. Most of the wrong assessments are simply due to interpreting minimized or packed JavaScript code as malicious. The false-negative, however, is due to "overlooking" a malicious curl command in the npm post-install hook of *delivery-promise 66.6.6*, which was correctly identified by text-bison.

There was an interesting edge case of wildly divergent ratings by text-bison for almost identical code snippets in different versions of the npm package *summa-wasm.* The only difference between versions 123.5 and 123.6 were name variations introduced when the component was packaged[5]. Text-bison rated one version as 9 (malicious) and the other as 0 (benign). Gpt-3.5-turbo demonstrated similar confusion, although not as extreme (2 versus 5).

The takeaway from this analysis is that both OpenAI gpt-3.5-turbo and Vertex AI text-bison produce very comparable assessments. This, however, does not mean that they perform well: both models produced a significant number of false positives, which would require manual review efforts and prevent automated notification to the respective package repository to trigger a package removal. That said, it does appear that models are improving. As described in previous blog posts [9, 10], gpt-4 produces significantly better results in regards to providing source code explanations and risk ratings for non-obfuscated code.

[5] These variations are probably due to a non-deterministic behavior of the minifier when it comes to replacing identifiers. For example, the same function is called *je* in one version and *ve* in the other one.

Those findings exemplify the difficulties of using LLMs for security-sensitive use cases. They can surely help manual reviewers, but even if assessment accuracy could be increased to 95% or even 99%, it would not be sufficient to enable autonomous decision making. The main impediment is the presence of educated and motivated adversaries who can patiently develop novel approaches that will remain undetected.

In addition to supporting defensive security efforts, LLMs are also very helpful to attackers creating malware, which underlines how they can be used for both good and evil. Thanks to their code comprehension and code generation capabilities, they can be used to alter malicious code such that it blends into a legitimate package targeted by the attacker. By selecting suitable identifiers, generating comments and distributing malicious code throughout the package, ChatGPT succeeds in hiding malicious code such that it consistently receives lower risk ratings from both LLMs.

However, to conclude this section on a more positive note, prompt injection is more manageable in this specific use case, because attackers do not live in a world entirely free of rules: their malware still needs to be valid code, which means that defenders can sanitize any code snippets included in a prompt. Comments should be removed and identifiers randomized such that the malware assessment is exclusively based on data or control flows, and not on human-readable content.

Call Graph Analysis

Reachability of Vulnerable Code

Takeaways:

- Simply understanding whether components contain vulnerabilities is not sufficient: 50% of analyzed Java projects that import vulnerable library versions do not reach any of the vulnerable code.
- The analyzed Java components use only 40% of vulnerable code detected in dependencies, suggesting significant opportunity to improve productivity in remediation efforts.
- The reachability of vulnerable components in tested projects varies significantly from 50-95% depending on the sample application and version. For example, Zookeeper v3.7.0 had only 58 of 209 reachable vulnerabilities, Lucene v9.4.0 had 2 out of 7 reachable, and Jenkins v2.307 had 57 of 78 reachable.

Census II is a collection of the most used open source Java projects in production applications operated by public and private organizations [11]. Last year's edition of this research report looked primarily at the dependency graphs of Census II components, as representative of typical programming practices across a wide variety of verticals and applications. We wanted to dig deeper this time and understand in greater detail how the code of those components interacts with the code of their dependencies - both at a given point in time, and also across different releases.

To this end, we created a comprehensive data set of more than 40,000 call graphs, which model reachability and help us understand function call patterns throughout the application [12]. They have been generated both for individual Maven components as well as projects with all dependencies, in which case the component-specific graphs are combined (or stitched) into a global call graph [13]. Such global call graphs were computed for all 6,113 releases of 307 different Census II components.

The analysis of all call graphs such as the ones shown in Figures 7 and 8 enables us to shed light on questions like the following:

- What are all the paths leading from developer-owned code to vulnerable methods in project dependencies?
- How much code do those Census II components import, how much of it is actually reachable, and how does this evolve across their releases?
- How many known vulnerable dependencies do the latest releases include, and what share of vulnerable methods is reachable in the context of the project code?
- Which security-sensitive APIs are used by Maven components, how does the use accumulate through dependency relationships, and how does that evolve across different releases?

Fig. 7

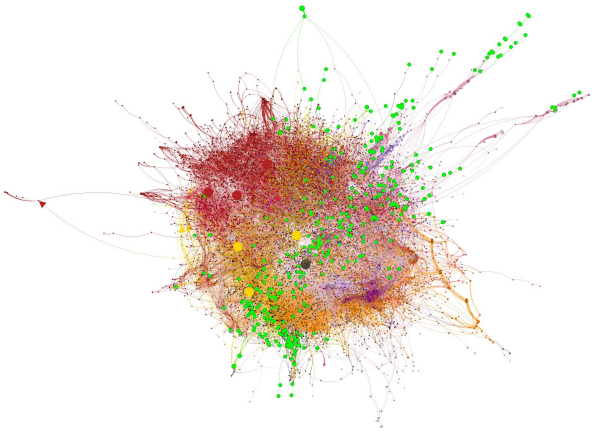

Fig 7 - A relatively small call graph with 14K+ nodes and 60K+ edges that represent all functions of logback-access v1.4.6 and its dependencies as well as function invocations.

Fig. 8

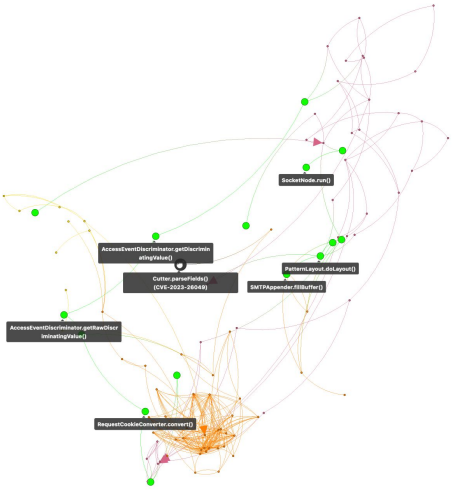

Fig 8 - Drill-down into paths leading from functions of logback-access v1.4.6 (in green) to the vulnerable function `CookieCutter.parseFields` affected by CVE-2023-26049.

To investigate the potential for static analysis to align remediation efforts with the risk presented by known vulnerabilities, we looked at the latest releases of 307 Census II components. Reachable vulnerabilities present much greater risk than unreachable ones, so our goal was to determine a "reduction rate" for each component: the proportion of known vulnerabilities that are not reachable.

We started by measuring the vulnerabilities reported in each project's Maven library identifiers and compile time dependencies [14], i.e. without considering any call graph information. Doing so shows that 15 components have exactly one vulnerability in their dependencies, 12 have two and so forth. In total, there are 277 vulnerabilities in 47 distinct Census II components, the majority of them in direct dependencies (197).

Then, for all of those vulnerabilities, we checked whether the vulnerable functions are actually part of the global call graph constructed for the respective Census II component. Across all projects, only 40% of vulnerabilities are reachable, which corresponds to a mean reduction rate of 60%.

A high standard deviation of 45% illustrates that the reduction rate varies significantly from one Census II component to the other (cf. Figure 9): In 24 out of 47 cases, none of vulnerable code was reachable, thus, the reduction rate was effectively 100%. In other cases, e.g., org.glassfish.jersey.media:jersey-media-json-jettison:2.3.9, the affected code of all its 5 vulnerabilities was part of the call graph.

Fig. 9

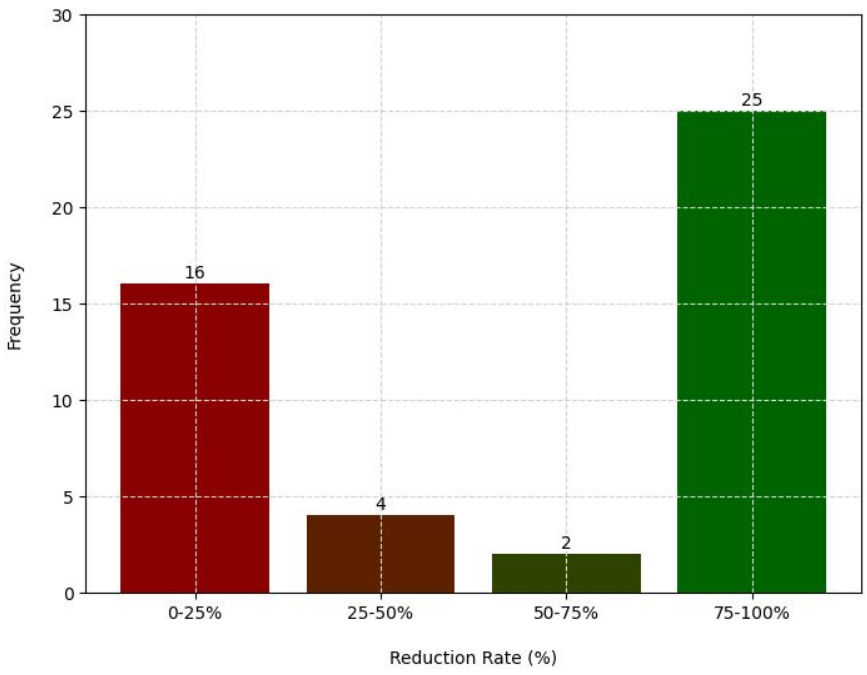

Fig 9 - Distribution of vulnerability reduction rate for 47 Java components from Census II: In 25 out of 47 cases, the reduction of vulnerabilities in compile and runtime dependencies is between 75-100%

Code Imported vs. Code Used

Takeaways:

- 71% of the code base of analyzed Census II components stems from imported OSS dependencies.
- Only 12-38% of imported OSS code is actually used according to call graph analyses.

By now, developers are very aware of dependencies and the security risks that accompany their use. Concepts such as direct dependencies and transitive dependencies are well understood[6], and it is common knowledge that modern applications depend on dozens, hundreds and sometimes thousands of 3rd party dependencies.

However, there's less visibility into the actual usage of all that code imported via dependencies. Some studies suggest that up to 75% of dependencies are unused and obsolete [15]. Using the dependency graphs and, more importantly, the call graphs of the Census II components, we set out to study code import[7] and code usage[8] at the granularity of lines of codes (LOC).

Figure 10 illustrates the percentage of imported third-party code in the total code base across all releases of the Census II components. The right-most bar, for example, shows that 1380 out of 3178 releases comprise between 90-100% of 3rd-party code, while the average rate of imported code is 71%.

[6] Direct dependencies are consciously selected and declared by developers, while transitive dependencies are the dependencies of direct dependencies that are automatically pulled into a project by a package manager.
[7] Third-party code distributed by means of packages, which are made available in a project workspace.
[8] Imported 3rd-party code that is part of the call graph of the respective Census II component.

Fig. 10

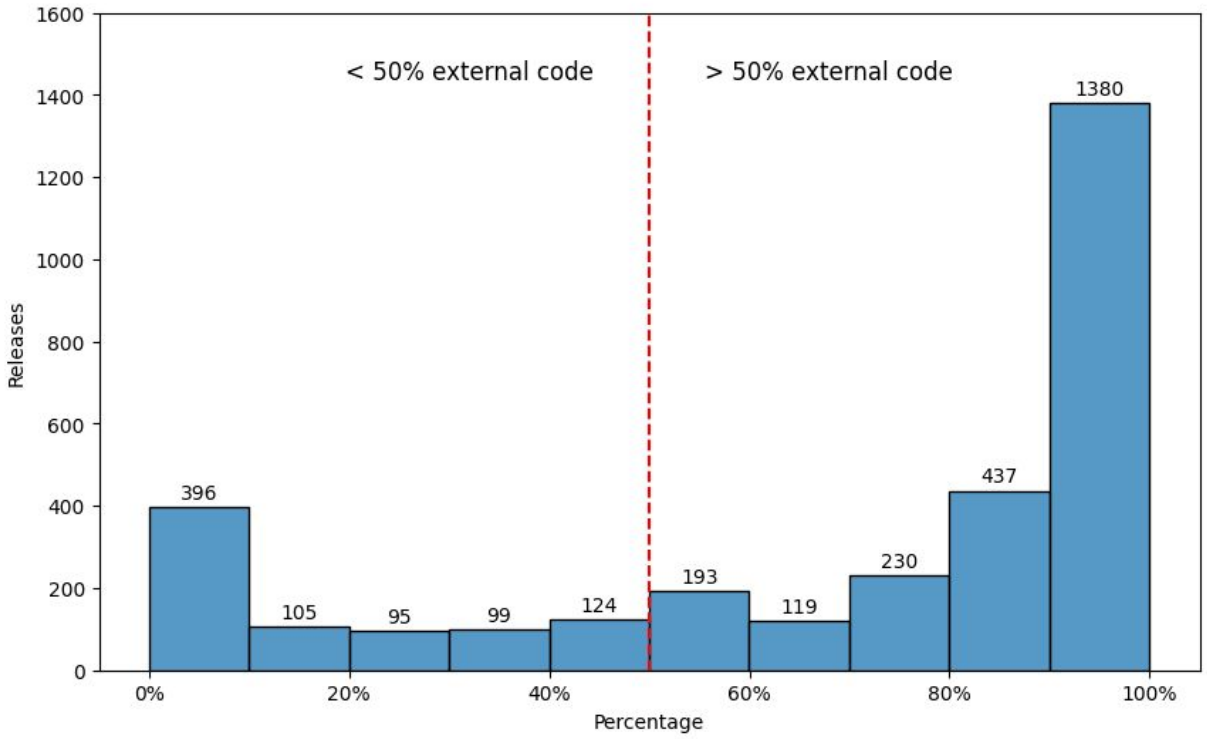

Fig 10 - Package composition: Percentage of imported code in total code base. The code base of 1380 releases, for example, comprises between 90-100% imported code.

If 71% of the code in Census II projects are from 3rd-party dependencies, on average, how much of that 3rd party code is actually used? We used the call graphs to determine which 3rd party code is called by functions of the Census II project.

Figure 11 presents the usage ratio of third-party LOC for direct and transitive dependencies. When comparing the plots, the usage rate for direct dependencies is generally higher than for transitive dependencies. For example, the number of releases that only use a small share of 0-5% of dependencies' LOC is much higher for transitive dependencies (604) than for direct ones (366). This is consistent with other research [15, 16], which also found most transitively resolved library dependencies remain unused.

Fig. 11

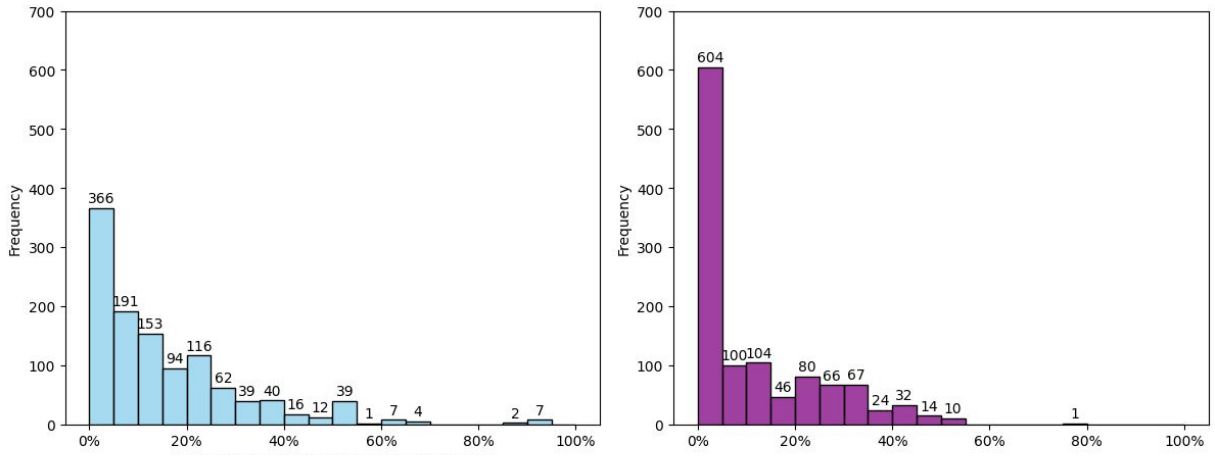

Fig 11 - Percentage of code in direct (left) and transitive (right) dependencies used: 366 Census II components, for instance, use only between 0-5% of the code in their direct dependencies.

In summary, across direct and transitive dependencies and considering all versions of Census II components, code reuse lies within the 12-38% range[9].

[9] When joining ("stitching") individual library call graphs, there is a need to account for ambiguous interface methods. For example, a call to List.add() in Java might end up calling ArrayList.add() or some other implementation of the List interface. We used a techniques called "no megamorphic call sites" which offers a representative picture of code use in real world applications. This technique excludes all interface calls that resolve to more than 5 targets from the callgraph [17] [18].

Security-sensitive Programming APIs

Takeaways:

- Components that use security sensitive APIs contribute to application risk, but developers lack visibility into such API use within those components.
- 71% of the analyzed Java components use 5 or more categories of security sensitive APIs directly or through its dependencies.
- 45% of the analyzed Java components inspect, load or execute code dynamically, mostly through Java reflection APIs (which can be problematic if user-provided input is not sufficiently sanitized, see for example CWE-95 or CWE-502).

All Java programs rely on the Java Class Library, which is a comprehensive set of standard classes providing a wide range of functionality such as basic numeric operations, graphical user interfaces and network-related functions.

Some of those classes provide read or write access to security sensitive resources such as the file system or the network interface. Those security sensitive APIs are typically fruitful targets for attackers. A path traversal vulnerability caused by insufficient input sanitization, for example, gives attackers access to file system resources outside of intended directories. Log4Shell, as another example, allowed attackers to load arbitrary Java classes with malicious payloads.

In other words, every use of a security sensitive API provides an opportunity for something to go wrong in case of a vulnerability. It doesn't matter whether the vulnerability occurs in the main application code or in a direct or transitive dependency. Reducing the number of times that sensitive APIs are used throughout the application is one way to reduce risk.

Developers, however, have very little insight into which security sensitive APIs are used by dependencies. As a result, they are unable to leverage this information when selecting components or assessing risk. To help address this gap, we have created eight categories of critical Java APIs and studied their use by components in the Census II data set. The category native-code, for example, contains APIs related to loading native system libraries.

Figure 12 shows the use of those sensitive API categories by 1758 individual components of the Census II data set. Interestingly, the category dyn-prg, which contains APIs related to reflection, script engines and URL class loaders, is the most-used category, used by 45% of all components. The use of such APIs has traditionally created problems for static analysis tools, but recent research suggests it is less of a problem for call graph analysis [19, 20].

Fig. 12

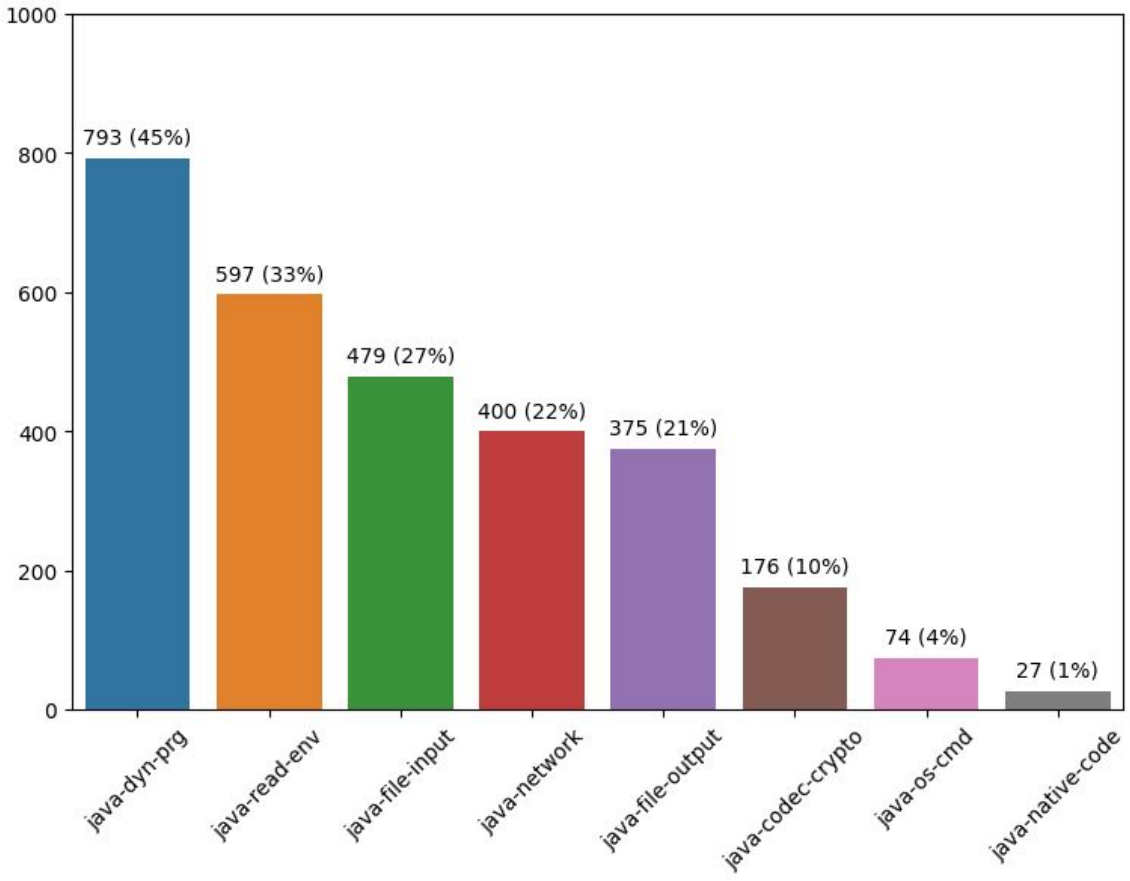

Fig 12 - Usage frequency of different sensitive API categories, computed over 1758 packages

Figure 13 illustrates that most components (45%) use no sensitive API category at all, and very few use all of them (8 out of 1758, e.g. the Web servers Apache Tomcat and Caucho Resin or Apache Hadoop).

However, Java components rarely exist in isolation. Typical Java applications contain dozens of dependencies and therefore accumulate sensitive API use. As shown by Figure 14, 71% of the analyzed components use 5 or more critical API categories if you include their dependencies in the analysis. For example, the most prominent logging libraries for Java, Log4j and Logback, use APIs from 6 out of the 8 sensitive API categories.

Fig. 13

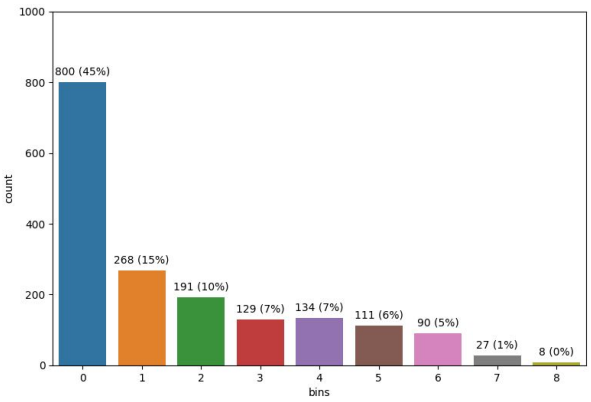

Fig 13 - Frequency of packages using none, one or multiple sensitive API categories (excluding dependencies), computed over 1758 packages

Fig. 14

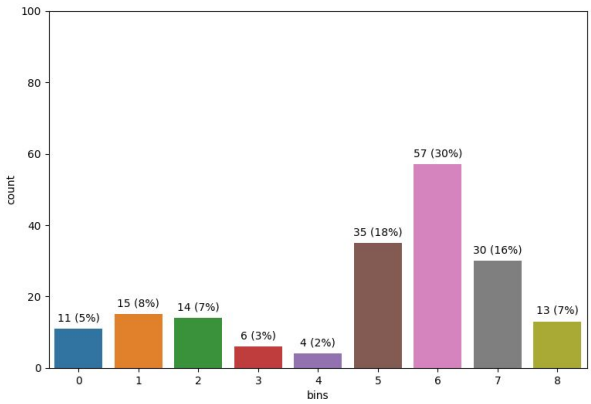

Fig 14 - Frequency of packages using none, one or multiple sensitive API categories (including dependencies), computed over 185 packages

In other words, even if your project does not use any of those sensitive APIs itself, the likelihood of being exposed is still relatively high when project dependencies are included.

If risk is creeping into applications through dependencies, and developers are updating dependencies over time, we also wanted to explore how their exposure might change over time. We compared the oldest and most recent releases of projects and determined that sensitive API related feature creep[10] does not seem to be a significant problem: 81% of the projects analyzed neither added nor removed sensitive API calls (cf. Figure 15).

[10] In computer science, feature creep is the on-going addition of new functionality in subsequent releases of a software product, which can result in software bloat and over-complication.

Fig. 15

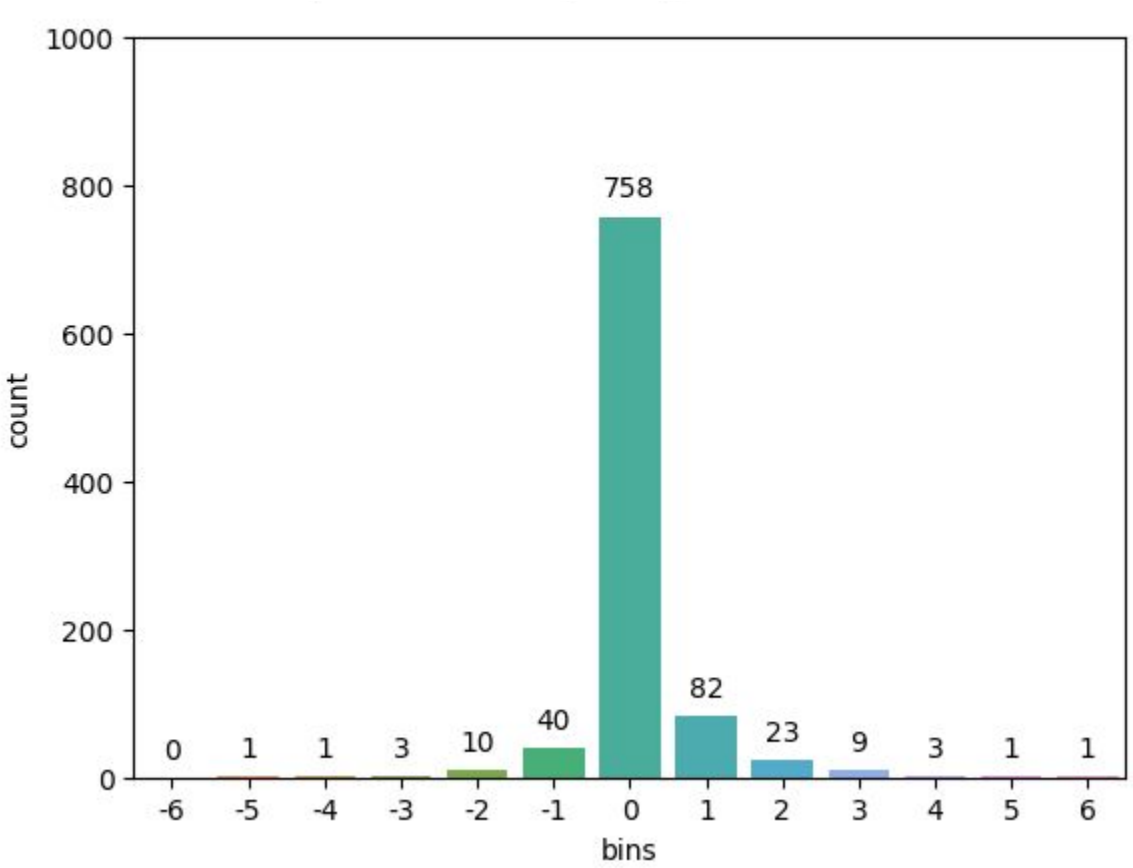

Fig 15 - Change of sensitive API use between first and most recent release, computed over 932 packages with > 3 releases

References

[1] https://www.unusual.vc/post/ai-native-infrastructure-will-be-open
[2] https://hazyresearch.stanford.edu/blog/2023-01-30-ai-linux
[3] https://www.semianalysis.com/p/google-we-have-no-moat-and-neither
[4] https://status.python.org/incidents/qy2t9mjjcc7g
[5] https://www.endorlabs.com/top-10-open-source-risks
[6] Garrett, Kalil, et al. "Detecting Suspicious Package Updates." *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, IEEE, 2019, pp. 13–16, https://doi.org/10.1109/ICSE-NIER.2019.00012.
[7] Ohm, Marc, et al. "On the Feasibility of Supervised Machine Learning for the Detection of Malicious Software Packages." *Proceedings of the 17th International Conference on Availability, Reliability and Security*, Association for Computing Machinery, 2022, pp. 1–10, https://doi.org/10.1145/3538969.3544415.
[8] Duan, Ruian, et al. *Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages*. 2, arXiv:2002.01139, arXiv, 2 Dec. 2020, https://doi.org/10.48550/arXiv.2002.01139.
[9] https://www.endorlabs.com/blog/reviewing-malware-with-llms-openai-vs-vertex-ai
[10] https://www.endorlabs.com/blog/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware
[11] H. Carter, "Census ii of free and open source software - lish/lf," Mar 2022. Available: https://data.world/thelinuxfoundation/ census-ii-of-free-and-open-source-software
[12] https://www.endorlabs.com/blog/what-is-reachability-based-dependency-analysis
[13] M. Keshani, "Scalable call graph constructor for maven," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 2021, pp. 99–101
[14] https://www.endorlabs.com/blog/what-are-maven-dependency-scopes-and-their-related-security-risks
[15] Soto-Valero, C., Harrand, N., Monperrus, M. et al. A comprehensive study of bloated dependencies in the Maven ecosystem. Empir Software Eng 26, 45 (2021). https://doi.org/10.1007/s10664-020-09914-8
[16] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," Empirical Software Engineering, vol. 27, no. 5, p. 102, 2022.
[17] Åkerblom, Beatrice, and Tobias Wrigstad. "Measuring polymorphism in Python programs." Proceedings of the 11th Symposium on Dynamic Languages. 2015.
[18] Agesen, Ole. "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism." ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9. Springer Berlin Heidelberg, 1995.
[19] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1049–1060.
[20] J. Liu, Y. Li, T. Tan, and J. Xue, "Reflection analysis for java: Uncovering more reflective targets precisely," in 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2017, pp. 12–23.

ENDOR LABS

Endor Labs helps developers spend less time dealing with security issues and more time accelerating development through safe Open Source Software (OSS) adoption. Our Dependency Lifecycle Management™ Solution helps organizations maximize software reuse by enabling security and development teams to select, secure, and maintain OSS at scale. The Endor Labs engineering team includes some of the world's leading static analysis experts, including 7 PhDs and senior engineers from Meta, Uber, Amazon, and Microsoft. Endor Labs was founded by industry veterans Varun Badhwar and Dimitri Stiliadis, and is backed by Lightspeed & Dell Technologies Capital, as well as executives at companies like Palo Alto Networks, Zscaler, Zoom, Google, and more