Measuring vulnerability to supply chain attacks and mitigating software dependency compromise

Derek Xu
dxu0117@bu.edu

Yongjoon Kweon
kweon10@bu.edu

Codebase: https://github.com/d0w/supplychainattack-research

I. Introduction

Supply chain attacks have emerged as a critical threat to the security of modern software and hardware systems. These attacks target the vulnerabilities of third-party components, libraries, and vendor services that organizations rely on for building and maintaining their systems. A compromised supply chain can lead to widespread security breaches, data leaks, and system downtime, thereby affecting not only individual organizations but also their customers and those related. With increasing software complexity increasingly relying on third-party packages and larger attack surfaces, ensuring the integrity of every component in the supply chain is both challenging and essential. A recent example involves the Go Module Mirror caching proxy service run by Google on behalf of Go developers which inadvertently hosted a backdoored package for over three years. This case along with countless others, underscores the critical importance of verifying the integrity of every component within a software supply chain. If one component of the software supply chain fails, then it can affect thousands of code repositories, compounding to further affect all further downstream codebases that depend on them.

Supply chain attacks involve a variety of attacks including upstream server, dependency confusion, open source software, or even phishing attacks. This paper plans to solve issues involving dependency and upstream server attacks. This involves mitigating the possibility of malicious code from being pushed to closed or open-source code repositories and also preventing downstream users that depend on the software from being affected as well. In the case of a compromised supply chain, we also need to mitigate issues that arise from deployed malicious software including backdoors or data leaks.

II. Problem Demonstration

To demonstrate the importance of supply chain security in regards to software dependencies, we start by looking at an attack surface of a simple project that uses third-party dependencies from package managers such as Pip and NPM. More specifically, we start by looking at the number of packages that are actually installed (as opposed to manually) for a given project. Using NPM, we are given a package-lock.json file that lists all the dependencies (of the dependency tree). Using Pip, we can view all the packages that are installed when we run the pip install command or with tools like pipdeptree. By installing the top 20 packages of pip or NPM one-by-one, we can count the number of actual installed packages based on the lockfile or the output of pip.

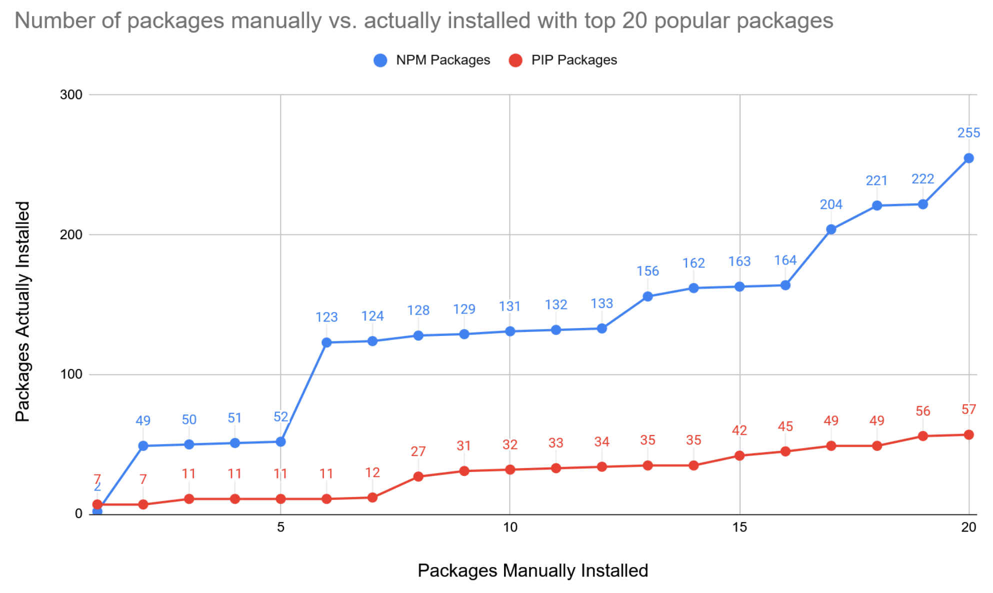
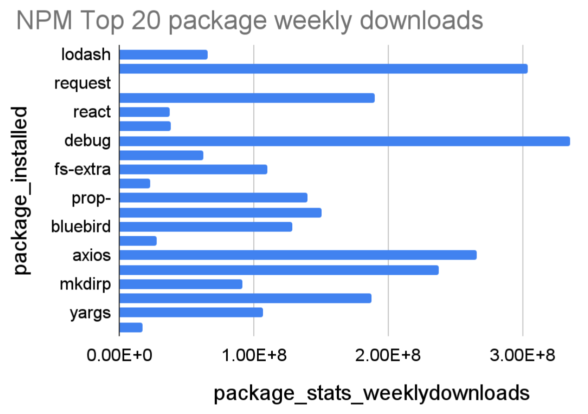


Figure 2.1 - The number of packages that are actually installed is much greater than the number that was installed by the developer.

With Figure 2.1, we see that with just 20 packages installed, NPM installed 255 actual packages. Pip, on the other hand, is slightly better at around 57. It's also useful to note that real-world numbers might be larger since the most installed packages are typically the most upstream dependencies, meaning they rely the least, but are relied on the most. The data demonstrates that even though a project might seem to have few dependencies, the true dependency tree is much larger and can greatly impact the attack surface of a codebase. In many ways, the utilization of third-party dependencies can immensely help developers create more advanced tools with abstracted libraries to choose from. Though, this comes at the cost of widening attack surfaces, leaving software with more points of vulnerability that are under no control of the downstream developer. Following the installation of just 20 packages, if any one of the 255 NPM packages of 57 pip packages are compromised or reveal a vulnerability, then a project utilizing the original 20 they installed is also subject to those same issues. Both the end-user of the vulnerable dependency, and all of the dependencies downstream are now at risk. Especially with well-known packages that may have many dependents, these issues can grow exponentially, affecting millions of users at once.

Next, we used the NPM and Pip registries to find how many users rely on these dependencies for their own projects or packages.



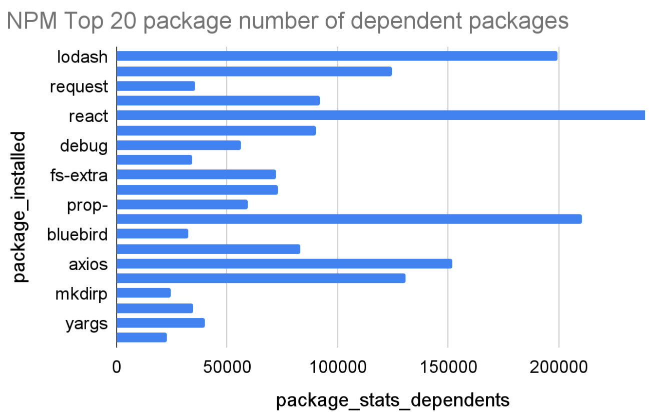


Figure 2.2 & 2.3 - Breakdown of top 20 NPM packages with their weekly download and dependent package counts (some packages are not shown for readability). Packages have many downstream codebases and users that rely on their code. Even if a package might not have many downloads, they can still have many dependents (i.e. React).

Package Manager	Avg Package Weekly Downloads	Avg Number of Dependent Packages
PIP	130,269,900	NA
NPM	132,445,833	90,724

Table 2.1 - Statistics for the top 20 installed packages of PIP and NPM package managers. Weekly numbers easily break 100 million users and the number of dependent packages averages nearly 100 thousand.

Package Name	**Last Week's Downloads (4/14/25)**	**Number of packages Required by (on example machine):**	**Another machine (Varsha)**
Numpy	99,425,715	125	96
Pandas	77,975,504	25	26
Matplotlib	19,982,226	1	11
Tensorflow	4,751,790	62	60

Table 2.2 - Case Study of the Statistics for an Average Machine of the Top Downloaded Packages on Python

As shown with Figures 2.2-2.3 and Table 2.1, packages can easily exceed hundreds of millions of downloads within a week's time frame. This along with a plethora of dependent packages that have their own sizable user bases, a vulnerability within any one of these packages could immensely impact the software security of entire nations. If, for example, a backdoor were implemented in Axios, roughly 250 million users within a week will run compromised code, and ~150,000 packages along with their respective users will inherit that backdoor. Although these are the top 20 downloaded packages in each registry, the concept is still the same. The number of users that can be impacted by the compromise of a single link in the software supply chain can be detrimental for much larger magnitudes of codebases due to how much dependencies rely on each other and how much they are used.

Furthermore, as illustrated in the case study presented in Table 2.2, many of the most frequently downloaded Python packages serve as dependencies for numerous other packages, often without the user's direct awareness. In our analysis, we examined the extent to which packages on our local machine depended on these top downloads. The results revealed that hundreds of installed packages indirectly rely on a small set of core libraries, highlighting the pervasive nature of software interdependencies. For instance, if a widely-used package such as NumPy were compromised, it would affect 125 other packages on our system alone, underscoring the potential magnitude of supply chain vulnerabilities.

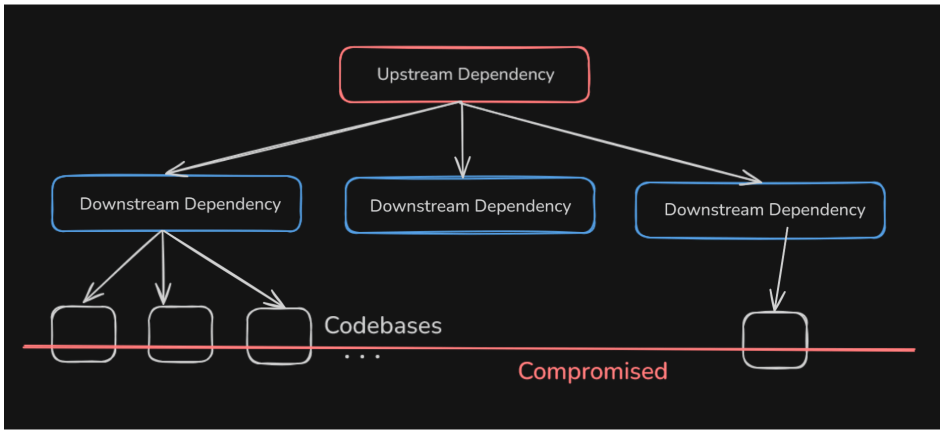


Figure 2.4 - Visualization of dependency tree. A singular compromised package can affect all software that lies downstream, affecting magnitudes more code than just a singular repository.

Expanding on the impacts of supply chain compromise, Figure 2.4 demonstrates just how much power one dependency can have on a multitude of dependencies and users that rely on it. Having just one repository along the software supply chain update with vulnerable code, such as backdoors or program injection scripts, can easily affect a large magnitude of users. Especially with large FOSS (free and open source) repositories with many contributors and limited maintainers, malicious code can slip past reviewers of pull requests if there are no automatic checks in place.

Trends in software engineering have prioritized the usage of many third-party libraries to achieve higher-level goals. Python and Javascript packages that use Pip and NPM for packages already demonstrate just how many dependencies are often used for tasks such as making API requests, creating a web server, or data analysis. Users will be abstracted away from code that may be made vulnerable with some form of supply chain attack. Especially with the sheer size of dependency trees, they can be prone to use outdated packages that have well-known vulnerabilities. Attackers need to simply probe this vulnerability across millions of repositories or inject their own malicious code into an existing codebase and can easily compromise systems due to how wide attack surfaces can become and just how impactful one broken link of a supply chain can become. All of these issues highlight the importance of cracking down on supply chain vulnerabilities, especially for both maintainers of popular repositories and the millions of downstream users that will deploy these packages to production-level servers.

III. Proposed Mitigations

VulScan (https://github.com/d0w/supplychainattack-research) serves as a suite of CLI tools that are meant to be used consistently like tests by users. The idea is for them to serve as hooks on actions such as Git commits, Git pull-requests, or updates to dependency folders. Our CLI tools will then run automatically on any of these triggers, and then produce a simple output that can be parsed by other tools or read directly by the developer. This output contains information on what files have potential security threats, the code under suspicion, dependencies that have known vulnerabilities, and a final severity score ranging from 0 (least vulnerable) to 10 (most vulnerable). This ideally gives programmers constant updates on the vulnerability of their code. If a new, vulnerable package is installed, VulScan will detect it and inform the user. If an existing package has a discovered vulnerability somewhere along its dependency tree, VulScan will detect it and inform the user. If a pull request into a codebase includes vulnerable code such as network requests, VulScan will detect it and inform the user. In general, the tools are meant to inform developers of their codebases' supply chains, and how its security changes with updates to the code or dependency lists. With this information, the effects of supply chain attacks can be mitigated since software engineers can easily generate vulnerability reports that can be integrated into CI/CD pipelines to prevent malicious code from being run on production-level servers with sensitive data, intercept vulnerabilities before they get downloaded by other users, etc.

VulScan has multiple tools. The first is the code-scanner which will take any folder and inspect all of the files within the folder recursively by statically analyzing the source code within. If given a folder, for every file, it will either string match for vulnerable patterns, generate and parse the AST (abstract syntax code), or a mix of both. For every flagged issue (such as protected file access or HTTP call), the vulnerability score of that file will be changed based on the severity of that vulnerability. These severities are values 0-10 that the end user can tweak for their own use cases. To complete this task, we create a set of analyzer scripts, each written to analyze the language that it's written in (analyzer.py for Python, analyzer.js for JavaScript, etc.). This is so we can use existing tools that are made for their respective language for tasks such as AST parsing. The CLI tool itself can use any language as its task is to open a directory, determine the correct analyzer to use, execute the respective script, and conglomerate results. VulScan uses Go for the main CLI tool for its extensive CLI libraries and concurrency capabilities. The tool will be able to run analysis on a variety of files in parallel execution which can help with dependency trees that can span thousands of files. Currently, VulScan's code-scanner is capable of analyzing for the following attack vectors, each with a severity weighting that can be altered.

We also created the GPT-scanner that leverages AI to help identify any supply chain vulnerabilities in our codebase. This is a tool that will take in a folder of codebases by simply traversing the path that is given through the program. This tool uses the power of AI and OpenAI's LLMs to detect vulnerabilities that might escape traditional pattern-matching or AST-based methods like we have above. Although we give the LLM the same metrics and patterns to look out for, it is still useful for finding and identifying context-dependent attacks. The tool works by first finding all of the files in the folder that is submitted for analysis by recursively traversing the directory structure. First, the scanner identifies the language that is used and then feeds the file into GPT-4o's OpenAI model. The prompt we used to feed into the LLM involved all of the metrics described above, and asked the model to also identify any suspicious activity that it saw. We also specified what the output of the model should look like so that it was consistent with the other analyzers (dependency and code-scanner). This meant that the analyzer came with quoted snippets of bad code and also a score of the risk level of each code file.

The goal of this tool is to have a better understanding of the intent of the code. There are millions of ways to write malicious code, and finding every scenario and keyword will simply be impossible if there is a malicious hacker who is trying to work their way around antivirus and anti-malware tools. VulScan also has a dependency-scanner tool that takes as input a requirements file such as requirements.txt or package.json, and checks for any known vulnerabilities. This helps for cases where source code for dependencies is not available or with security issues that require much more nuanced understanding that AST and regex parsing cannot provide. Generally, this means that this tool will provide more accurate information on complicated vulnerabilities, but cannot mitigate zero-day attacks, where an exploit is used before it is recognized by anyone that can mitigate it.

IV. Solution Results

To test the effectiveness of our CLI tools, we used both handwritten dummy codebases as well as installed NPM and Python packages with known vulnerabilities, and logged the output of VulScan. For handwritten codebases, we deliberately wrote in vulnerable lines of code to see if VulScan finds those malicious snippets. With known vulnerable packages, our dependency scanner will scan against requirement files that we intentionally add vulnerable packages found from online repositories into.

Codebase	**Number of Vulnerabilities Inserted**	**No. of Vulnerabilities Detected - Code-scanner (score)**	**No. of Vulnerabilities Detected - GPT-Scanner (score)**
Python Insecure Codebase	46	48 (10.0)	32 (10.0)
Python Secure Codebase	0	4 (2.0)	0 (0.0)
Python Intent-based Codebase	4	1 (2.0)	4 (8.25)
JavaScript Insecure Codebase	30	34(10.0)	7(9.20)

JavaScript Secure Codebase	0	1(1.0)	5(2.0)

Table 4.1 - Statistical analysis of 3 different dummy codebases with vulnerabilities and their results against our static analysis tools

Our research evaluated three Python and 2 Javascript codebases using two different vulnerability scanning approaches: a traditional pattern-matching analyzer (code-scanner) and our GPT-based semantic analyzer (GPT-Scanner). The results revealed distinctive strengths and limitations of each approach. In our deliberately vulnerable codebase containing 46 security flaws, Code-Scanner identified 48 vulnerabilities (scoring 10.0), slightly overreporting due to its pattern-matching methodology that occasionally flags similar syntax patterns. It was also able to score the highest vulnerability rate(10.0) for the Javascript codebase. The GPT-Scanner detected 32 vulnerabilities (scoring 10.0) in the Python insecure codebase, and 7 vulnerabilities in the Javascript insecure codebase (scoring 9.2).

The most revealing insights came from our intent-based codebase, which contained 4 sophisticated python vulnerabilities designed to evade traditional scanning. Here, Code-Scanner identified only 1 vulnerability (scoring 2.0), while the GPT-Scanner detected all 4 (scoring 8.25). This demonstrates GPT-Scanner's superior ability to understand the code's context and developer intent rather than just syntax patterns. For example, our intent-based codebase used another socket variable "s" instead of the typical "socket.socket()" pattern, successfully evading Code-Scanner. Similarly, we implemented SSH private key exfiltration using non-standard access patterns that bypassed traditional detection rules. The GPT-Scanner recognized both the undefined variable issue and the sensitive credential access despite their obfuscated implementation. Metamorphism can severely hinder a tool's ability to catch every single vulnerability present in the code as there are a million different ways that an attacker can do the exact same thing. If an attacker wanted to, they could even look for ways to bypass our Code-Scanner in other ways such as putting a check to something unrelated to simulate "good" coding.

Despite its strong contextual understanding, the GPT-Scanner exhibits several significant limitations. When analyzing large codebases, the model frequently truncates files to fit within its context window, compromising its detection capabilities. This is evident in its 66.6% (Python) and 23.3%(Javascript) detection rate on our comprehensive insecure codebases. While increasing the token limit could theoretically address this issue, doing so would substantially increase processing time, rendering the scanner impractical for routine security workflows.

Real Codebase	**Pip/NPM Audit**	**Dep-Scanner Score**	**Code-Scanner Score**
JS FullStack Project with outdated packages	3 moderate, 2 high	8	7.2
~ with up-to-date packages	0	0	2.7
Python Web Backend	4 moderate, 1 high	7.5	10.0

with outdated packages			
~ with up-to-date packages	0	0	3.5

Table 4.2 - Comparison of package manager audit, dependency scanner, and code scanner scores across different usage scenarios with vulnerable packages installed.

To test against real codebases, we create a variety of projects that use well known packages for tasks such as API or SPA (Single page application) development. With these projects, we run an audit using a known auditing tool such as npm audit or pip-audit, and also run our dependency and code scanners. We see that with moderate to high audit issues, our scanners also produce a vulnerability score accordingly. A project with 3 moderate, and 2 highly vulnerable packages, has a dependency scanner score of 8, and code scanner score of 7.2. It is useful to note the limitations of the code scanner as being variable to a developer's needs. Since the severity weights for the score can be configured by users for their own needs, the code scanner score can vary drastically from project to project. Furthermore, since pip often installs binaries, we were unable to scan some of the dependencies, leading to a slightly misfigured score. On the other hand, since the dependency scanner compares our requirements file with a list of known vulnerabilities, it is clear that this solution is not perfect or extremely accurate either. Even when testing amongst known vulnerable packages, our scanner will not recognize the package version as creators may delete these packages, or a vulnerable package may not be identified because the vulnerability has not been discovered yet. Overall, with both dependency and code scanners combined, it can cover a wider range of attack vectors since together they have the ability to cover for zero-day attacks with the static analysis as well as check for existing vulnerabilities when source code is not available.

Although results from our dummy codebases and real usage scenarios may not be completely accurate and represent the actual vulnerability of a codebase, they show that with an increased use of vulnerable packages, we get an increased vulnerability score from VulScan which points in the right direction for inhibiting the effects of supply chain attacks.

V. Future Work and Conclusion

Securing the software supply chain is no longer optional—it is essential. As demonstrated through real-world case studies and our experiments with VulScan, even a single compromised dependency can cascade into widespread vulnerabilities affecting thousands of downstream projects and millions of users. Our tools aim to make supply chain security more accessible, providing developers with actionable insights through both static and semantic analysis. While no solution is foolproof, combining code analysis with dependency tracking significantly reduces risk. When used in conjunction with other tools, VulScan can let users monitor the security of their codebases while automating security checks and actions. Future improvements, such as CI/CD integration and dynamic analysis, will enhance coverage and responsiveness, strengthening the resilience of modern software systems against evolving threats.

Contributors

Christopher Hyun (chyun121@bu.edu) - Javascript codebase testing (Table 4.1)

Eugene Seoh (eseoh@bu.edu) - Javascript codebase testing (Table 4.1)

References

Trend Micro
https://www.trendmicro.com/en_us/research/23/j/infection-techniques-across-supply-chains-and-codebases.html

Crowdstrike
https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/supply-chain-attack/

Node Package Manager
https://www.npmjs.com/

Python Package Index
https://pypi.org/

Python Package Index Stats
https://pypistats.org/top

npm-rank - Andrei Kashcha
https://gist.github.com/anvaka/8e8fa57c7ee1350e3491

Safety Database - Safety Cybersecurity (Pyup.io)
https://github.com/pyupio/safety-db/blob/master/data/insecure.json

NPM Vulnerability Repository - Liang Gong, Nicholas Starke
https://github.com/JacksonGL/NPM-Vuln-PoC