

# Open Source, Open Threats? Investigating Security Challenges in Open-Source Software

Seyed Ali Akhavani\*  
Northeastern University  
Boston, MA, USA  
sadatakhavani.s@northeastern.edu

Behzad Ousat\*  
Florida International University  
Miami, FL, USA  
bousat@fiu.edu

Amin Kharraz  
Florida International University  
Miami, FL, USA  
ak@cs.fiu.edu

\*Authors contributed equally to this work.

**Abstract**—Open-source software (OSS) has become increasingly more popular across different domains. However, this rapid development and widespread adoption come with a security cost. The growing complexity and openness of OSS ecosystems have led to increased exposure to vulnerabilities and attack surfaces. This paper investigates the trends and patterns of reported vulnerabilities within OSS platforms, focusing on the implications of these findings for security practices. To understand the dynamics of OSS vulnerabilities, we analyze a comprehensive dataset comprising 31,267 unique vulnerability reports from GitHub’s advisory database and Snyk.io, belonging to 14,675 packages across 10 programming languages.

Our analysis reveals a significant surge in reported vulnerabilities, increasing at an annual rate of 98%—far outpacing the 25% average annual growth in the number of open-source software (OSS) packages. Additionally, we observe an 85% increase in the average lifespan of vulnerabilities across ecosystems during the studied period, indicating a potential decline in security. We identify the most prevalent Common Weakness Enumerations (CWEs) across programming languages and find that, on average, just seven CWEs are responsible for over 50% of all reported vulnerabilities. We further examine these commonly observed CWEs and highlight ecosystem-specific trends. Notably, we find that vulnerabilities associated with intentionally malicious packages comprise 49% of reports in the NPM ecosystem and 14% in PyPI—an alarming indication of targeted attacks within package repositories. We conclude with an in-depth discussion of the characteristics and attack vectors associated with these malicious packages.

**Index Terms**—Supply Chain Security, Open-Source Security

## 1. Introduction

The open-source software (OSS) landscape has seen exponential growth over the past decade, becoming a backbone for a wide range of industries, including aerospace [1], energy systems[2], [3], finance [4], [5],

healthcare [6], and government projects. As the adoption of OSS starts to dominate across different sectors, its inherent benefits—such as transparency, collaboration, and rapid innovation—are accompanied by important challenges. The open and decentralized nature of OSS, while promoting collaboration and innovation, also creates opportunities for adversaries to exploit vulnerabilities [7], compromise supply chains [8], [9], and introduce malicious code into widely-used libraries [10], [11], [12].

Government entities, policymakers, and regulators recognize the need for stronger security measures in the open-source environment. The recent United States *Securing Open Source Software Act* represents a significant step toward mandating security improvements and creating frameworks for the protection of open-source libraries and dependencies [13], [14]. Similarly, United States’ Cybersecurity and Infrastructure Security Agency (CISA) has published the *Open Source Software Security Roadmap*, outlining its strategy to ensure a secure open-source ecosystem [15]. These initiatives reflect growing concerns over the reliance on OSS in critical infrastructure, highlighting challenges like dependency visibility, large-scale vulnerability management, and human errors in maintaining packages.

Our work is guided by three primary research questions. 1) How have vulnerabilities evolved in different ecosystems?, 2) How long do vulnerabilities persist and spread across packages?, 3) What are the unique vulnerability patterns in different ecosystems? To answer these questions, we generated a dataset of vulnerability reports from 2017 to January 2025. The data consists of over **31,267 reported vulnerabilities** from two advisory databases—GitHub Advisory Database and Snyk.io— that cover **14,675 open-source repositories** publicly available on GitHub. We also have reports of **4,456 malicious packages** whose source code is no longer available since they have been permanently removed. We examine vulnerabilities across **10 programming languages**—C, C++, PHP, Rust, JavaScript (Node.JS), Go, Java, .Net, Python, Ruby—, and **eight**

**distinct package managers**—PyPi, NPM, Composer, Crates, Go, Maven, NuGet, and RubyGems— which are foundational to software and web development [16], [17], [18], [19]. This longitudinal analysis allows us to track the growth and distribution of vulnerabilities across programming languages.

In the following, we highlight some of the major findings of this paper.

**The rapid adoption of open-source platforms is being overshadowed by a faster rise in reported vulnerabilities.** The analysis reveals a substantial increase in the number of packages across open-source platforms, with an average annual growth rate of 25.16%. However, this growth is accompanied by an alarming rise in reported vulnerable packages, which have increased at a much faster rate—98% annually. This suggests that as package managers rapidly expand their number of packages, the security risks associated with them are growing disproportionately. For example, in 2022, there were 2,490 vulnerabilities reported for Maven and 1,593 for Composer. The sharp increase in vulnerabilities, particularly in high-growth platforms, highlights a critical security gap, as the rate of vulnerabilities is outpacing the platform’s expansion, making open-source software an increasingly attractive target for attackers.

**The expanding scale of open-source ecosystems does not necessarily translate into improved security.** We analyze vulnerabilities at the package level and find that the time a vulnerability remains in the ecosystem before being fixed has increased by 95% from 2017 to 2024. Additionally, we examine vulnerability concentration patterns to assess whether security issues stem from a small number of packages or are dispersed across the ecosystem. Our analysis reveals distinct trends—some platforms exhibit a high concentration of vulnerabilities in key packages, while others show a broader, more distributed spread. For instance, Composer (4.92 vulnerabilities per vulnerable package) and C/C++ (3.96) have vulnerabilities concentrated in a few critical packages. In contrast, ecosystems such as NPM (1.75 vulnerabilities per package) and Go (1.70) show a more even distribution of vulnerabilities.

**The evaluation of CWEs across ecosystems reveals distinct vulnerability patterns shaped by both common threats and language-specific characteristics.** Our results reveal key trends in vulnerability distribution, highlighting the prevalence of certain vulnerabilities and the impact of language-specific characteristics. Our data demonstrates that, on average of the investigated platforms, only seven CWEs were responsible for over 50% of the vulnerability reports. Notably, CWE-79 (Cross-Site Scripting) and CWE-22 (Path Traversal) pose a major threat in ecosystems like Composer, NPM, PyPI, and Maven, reflecting widespread security challenges in web applications. On the other hand, some CWEs are deeply tied to language characteristics, such as CWE-122 (Heap-based

Buffer Overflow) in C/C++ or CWE-1321 (Prototype Pollution) in JavaScript. We specifically focus on CWE-506 (Embedded Malicious Code) that has emerged as a significant issue in platforms like NPM and PyPI, where 48.58% and 13.92% of all vulnerabilities, represent intentional attacks underscoring the growing concern over supply chain attacks.

We hope this work to raise awareness about the increasing need for systematic approaches to enhance visibility in open-source software. We also hope that the insights provided in this paper will inspire new directions in developing automated tools that can effectively model vulnerabilities and augment developers’ experience to build more robust code and test cases. We summarize our contributions as follows:

- We present a longitudinal study that explores open-source software package across 10 programming languages and eight distinct package managers from 2017 to 2025, measuring the growth rate of packages and vulnerabilities.
- We investigate the lifespan of vulnerabilities over the last years across the ecosystems and discuss the uneven distribution of vulnerabilities across packages, identifying ecosystems with concentrated or widespread risks.
- We discuss the vulnerability types that are common across multiple platforms and others that are specifically reported for an ecosystem, requiring specific mitigation strategies for each case.
- We provide a detail analysis of intentionally malicious packages observed mostly in NPM and PyPI ecosystems, showcasing their characteristics and attack vectors.

**Availability.** We provide the research artifacts, including datasets and analysis scripts for further research <sup>1</sup>. This dataset serves as a valuable resource for future research in the field, as no publicly available dataset includes such valuable information in a unified format.

## 2. Background and Related Work

In this section, we provide a background on the vulnerability disclosure process and describe three main aspects of the security and trustworthiness of OSS projects. In particular, we discuss dependency networks, dependency network vulnerabilities, and package-level vulnerabilities. We then discuss our work, focusing on the longitudinal data about reported vulnerabilities and emerging trends in OSS over the past years. Table 1 presents the focus of related studies compared to our work. In the following, we describe different categories and briefly discuss the contributions of each study.

### 2.1. Vulnerability Disclosure Process

The vulnerability reporting process is vital for security management, involving key systems to identify

1. <https://github.com/sa-akhavani/oss-security>

TABLE 1. OVERVIEW OF RESEARCH PAPER DETAILS ON PACKAGE VULNERABILITY AND DEPENDENCY NETWORK ANALYSIS.

Category	Reference	Total Vulnerabilities	Total Packages	Evaluation Period	Package Managers									
					PyPi	NPM	RubyGems	Maven	Cargo	Packagist	NuGet	Go	CPAN	CRAN
Package Vulnerability	[20]	1,396	2,224	2006-20	✓									
	[21]	609	-	2011-18		✓								
	[22]	339	1M	n/a	✓	✓	✓							
	[23]	5,916	958,547	NA-2022	✓	✓								
Dependency Network	[24]	0	449,518	2011-16		✓	✓							✓
	[25]	0	830,000	2012-17		✓	✓		✓	✓	✓		✓	✓
	[26]	0	253,896	2005-16		✓	✓		✓					
Dependency Vulnerability	[27]	400	610,000	2012-17		✓								
	[28]	2,874	867,290	2011-20		✓	✓							
	[29]	n/a	200	n/a				✓						
Package Vulnerability	Our Study	<b>31,267</b>	<b>14,675</b>	<b>2017-25</b>	✓	✓	✓	✓	✓	✓	✓	✓		

and classify issues. Common Vulnerabilities and Exposures (CVE) [30], managed by MITRE [31], is a reference list of publicly known vulnerabilities, each assigned a unique identifier. Complementing CVE, Common Weakness Enumeration (CWE) [32] serves as a community-developed taxonomy for identifying software weaknesses. Each weakness in the CWE list is assigned a unique identifier, simplifying the tracking and addressing of these issues across platforms. Lastly, the National Vulnerability Database (NVD) [33] enhances CVE data with additional information, improving the context and understanding of vulnerabilities.

## 2.2. Dependency Network and Vulnerability Analysis

Numerous studies explore the structure and evolution of dependency networks in open-source projects. Building on this, other research investigates vulnerabilities within these dependency chains. We categorize these studies into two groups: *Dependency Network* and *Dependency Vulnerability* analyses. In the following, we will describe each category in more detail.

**Dependency Network Studies** Decan et al. [24], compare the dependency graphs across NPM, CRAN, and RubyGems, analyzing 449,518 packages from 2011 to 2016. They identify structural issues related to transitive dependencies and compare how these issues evolve over time across different ecosystems. Decan et al. [25] further explore this and examine the evolution of dependency networks in Cargo, CPAN, CRAN, NPM, NuGet, Packagist, and RubyGems between 2012 and 2017, analyzing approximately 830,000 packages. The authors investigate how the structure and complexity of these networks change over time, providing insights into the growth and maintenance challenges of different ecosystems. Kikas et al. [26], study the NPM, RubyGems, and Crates ecosystems from 2005 to 2016, analyzing 253,896 packages. They explore the structural properties and evolution of package dependency networks, highlighting differences and similarities among ecosystems and discussing implications for dependency management tools.

**Dependency Vulnerability Studies** Decan et al. [27], present an empirical study of nearly 400 security reports over a six-year period in the NPM ecosystem, which includes over 610,000 JavaScript packages. The authors analyze how and when vulnerabilities are discovered and fixed, and the extent to which they affect other packages within the ecosystem, considering the severity of vulnerabilities and the presence of dependency constraints. Zerouali et al. [28] conduct an empirical analysis of vulnerabilities reported in the NPM and RubyGems ecosystems between 2011 and 2020, covering 2,874 vulnerabilities in 867,290 packages. They examine how vulnerabilities propagate through dependency networks and assess the impact on both direct and transitive dependents, providing insights into the security risks associated with package dependencies. The Pashchenko et al. [29] study focuses on the Maven ecosystem, analyzing the risks associated with vulnerable dependencies. The authors propose methods to assess the actual impact of vulnerabilities by considering factors such as usage popularity and the presence of alternative safe versions, aiming to provide actionable information for software development companies regarding their library dependencies.

Our study does not focus on the dependency networks of vulnerable packages, as prior research has extensively covered this area. Instead, we focus on the vulnerabilities and CWEs across ecosystems, which aligns with the scope of our work. Dependency chain analysis, while valuable, requires an in-depth investigation beyond our current objectives.

## 2.3. Package Vulnerability Analysis

Several studies have examined vulnerabilities' nature, frequency, and impact in different ecosystems. This subsection highlights notable research efforts investigating the identification, evaluation, and trends of security vulnerabilities in popular ecosystems.

Alfadel et al. [20] conducted an empirical study of 1,396 vulnerability reports affecting 698 Python packages in the PyPi ecosystem, covering the period from 2006 to 2020. Their analysis revealed that the discovery

and remediation times for vulnerabilities vary significantly, with some vulnerabilities remaining unaddressed for extended periods. The study also highlighted that a substantial number of vulnerabilities are introduced in early versions of packages and persist across releases.

Zimmermann et al. [21] examined the NPM ecosystem, focusing on its dependency network and associated security risks. Analyzing 609 publicly known security issues from 2011 to 2018, they found that the connected nature of NPM introduces several weak spots. Specifically, installing an NPM package introduces an implicit trust in numerous third-party packages and maintainers, creating a surprisingly large attack surface. The findings suggest that the interconnectedness of the NPM ecosystem can lead to widespread security vulnerabilities.

Zahan et al. [23] investigated the correlation between software security practices and the prevalence of vulnerabilities in open-source packages across the PyPI and NPM ecosystems. Analyzing 5,916 vulnerabilities affecting 958,547 packages up to 2022, they assessed various security measures, such as the use of automated tools, adherence to security guidelines, and community engagement. Their study concluded that packages following robust security practices tend to have fewer reported vulnerabilities, highlighting the effectiveness of proactive security measures in reducing risks. Zahan et al. [34] continue the analysis of metadata of 1.63 million NPM packages and proposed signals of security weaknesses.

Our study provides a comprehensive view of vulnerabilities, investigating ten programming languages, a scope unmatched by any prior work, which have typically focused on analyzing a smaller number of ecosystems. Notably, none of the previous studies have included an analysis of C and C++ package vulnerabilities, a gap that our research addresses. Additionally, the vulnerability report datasets analyzed in earlier works are significantly smaller. We analyze 31,267 vulnerability reports from 14,675 packages, making our vulnerability report dataset five times larger than the existing vulnerability analysis in prior studies. We have listed the prior work in Table 1 to demonstrate what has been the focus of open-source ecosystem study in the past and what differentiates our work from previous studies.

### 3. Research Questions

We focus on vulnerabilities reported until 2025, including recent data points for open-source security due to significant shifts and increasing reported CVEs. Most prior studies focus on data from 2018 or earlier. However, we have seen substantial changes in vulnerability patterns in 2022, alongside a significant increase in the number of published packages and reported vulnerabilities. Our analysis addresses these gaps, providing a timely and necessary exploration of new trends and concerns. We address three main research questions that drive our study, designed to uncover key insights into

open-source security and provide actionable findings for developers and security practitioners.

**RQ1: How have vulnerabilities evolved over time in different ecosystems?** Identifying emerging flaws and vulnerabilities is critical to the development and testing of modern defense solutions such as fuzzing tools, vulnerability scanners, and program analysis techniques. This question focuses mainly on trends in open-source software packages and vulnerabilities in published packages across different ecosystems. We investigate how the introduction of new packages across different environments has contributed to the emergence of specific forms of vulnerability. We examine how the introduction of new packages and the overall growth in ecosystems (e.g., NPM, PyPI, Maven) contribute to the emergence of vulnerabilities. We look at Common Vulnerabilities and Exposures (CVEs) compared to the overall growth in the number of packages.

*Key Finding:* Our analysis reveals that the growth rate of vulnerabilities significantly outpaces package growth across ecosystems, with an average annual increase of 98% in vulnerable packages compared to 25% growth in total packages, indicating a deteriorating security posture in open-source ecosystems.

**RQ2: How do vulnerabilities persist and concentrate across different ecosystems?** Understanding the distribution of vulnerabilities provides useful insights into ecosystem security dynamics. This research question examines two critical aspects: How long vulnerabilities persist in packages before being addressed (vulnerability lifespan), and how vulnerabilities are concentrated within ecosystems. We analyze vulnerability lifespan trends to understand whether improved community engagement is the reason behind the growth in the number of reported vulnerabilities or not. Additionally, we investigate vulnerability concentration patterns to determine whether security issues are seen in a few critical packages or distributed broadly across ecosystems. This analysis helps identify whether certain packages become repeated targets and informs risk assessment strategies for dependency management.

*Key Finding:* Despite the increasing number of vulnerability reports over time, our study reveals a critical trend: vulnerability lifespans are increasing across all platforms, with an average of 85% increase for the duration of the study across different ecosystems. This indicates that while more vulnerabilities are being discovered, they persist longer in the software supply chain. Furthermore, we observe significant variations in vulnerability concentration, with ecosystems like Composer showing high concentration (4.92 vulnerabilities per vulnerable package) versus NPM showing broader distribution (1.75 vulnerabilities per package).

**RQ3: What are the distinct vulnerability patterns and attack vectors across ecosystems, particularly regarding malicious packages?** Different ecosystems exhibit unique vulnerability characteristics shaped by their design principles and developer practices. This

TABLE 2. OVERVIEW OF THE COLLECTED VULNERABILITY REPORTS FROM GITHUB ADVISORY AND SNYK DATABASE PER ECOSYSTEM.

Platform	Snyk	GitHub Advisory	Unique Reports	Affected Packages
C/C++	1719	-	1,719	383
Composer (PHP)	1,203	4,698	4,450	788
Crates (Rust)	695	1,015	1,435	575
Go	1,363	2,533	1,657	
Maven (Java)	1,280	6,036	5,847	2,455
NPM (Node.JS)	4,740	4,248	8,237	6,245
NuGet (.Net)	947	663	1,312	581
PyPI (Python)	1,265	3,902	3,907	1,614
RubyGems (Ruby)	407	944	1,191	377
Total	13,828	20,968	31,267	14,675

research question explores CWE distribution patterns to identify both common threats and ecosystem-specific vulnerabilities. We analyze the prevalence and trends of different CWEs across major ecosystems from 2017 to 2025, investigating which vulnerabilities are observed in multiple languages and which are specific to individual ones. Additionally, we examine the emerging threat of intentionally malicious packages (CWE-506), which represents a fundamental shift from accidental vulnerabilities to deliberate attacks. We investigate the distribution, attack vectors, and evolution of malicious packages to understand this growing supply chain threat.

*Key Finding:* Our analysis reveals that while certain vulnerabilities like Cross-Site Scripting (CWE-79) and Path Traversal (CWE-22) are observed across different ecosystems, some are deeply tied to language characteristics, such as memory management issues in C/C++ or prototype pollution in JavaScript. Most significantly, we identify an alarming concentration of malicious packages (CWE-506) in NPM and PyPI, where 48.58% and 13.92% of all vulnerabilities represent intentional attacks rather than accidental flaws.

## 4. Data Collection

The pipeline integrates two primary data sources of reported vulnerabilities. GitHub Advisory Database [35], and Snyk.io [36]. These databases are particularly relevant to the research questions due to their structured and detailed categorization of vulnerabilities. Additionally, we use Libraries.io [37] and GitHub.com [38] to collect metadata from the source code and retrieve the information for each repository. Below, we describe each data source, then we demonstrate how we integrated all the gathered information into a comprehensive queryable data source for our research evaluation.

### 4.1. Vulnerability Reports

**GitHub Advisory Database.** The GitHub Advisory Database [35] is a publicly accessible repository that provides security advisories for open-source software projects. This advisory uses the Open Source Vulnerability format (OSV) [39] to generate its vulnerability reports. This format includes information such as the

package name, ecosystem, severity, affected versions, and remediation steps, as well as references to CVE and CWE identifiers. The Advisory Database offers two distinct datasets for security incidents, covering data from 2017 to the present. *GitHub-reviewed advisories*, which at the time of this study, include over 21,000 reports. And *Unreviewed advisories*, containing more than 242,000 reports. The GitHub-reviewed advisories are security vulnerabilities that have been mapped to packages in the supported languages. GitHub reviews each advisory in this dataset for validity and ensures that they have a full description, and contain both ecosystem and package information. This comprehensive data might not be available for the unreviewed reports. In this study, we only collected the data on the reviewed advisories to ensure that we work on reliable data. The full advisory dataset is provided by GitHub in a public repository [35]. We extracted a total of 17,356 records from the advisory database from eight different ecosystems for our studied programming languages.

**Snyk.io** Snyk Vulnerability Database [36] is another well-supported repository of security vulnerabilities for open-source projects. Snyk database aggregates data from its own disclosed vulnerabilities, in addition to other public sources such as the National Vulnerability Database (NVD), community contributions, and its proprietary research team. Snyk specializes in identifying vulnerabilities across a wide range of ecosystems, including all the ones in our study, providing detailed information such as vulnerability descriptions, ecosystems, affected versions, and severity ratings. Prior work has utilized Snyk database for code analysis[40], [41], Docker vulnerability analysis [42], and development of vulnerability mitigation tools [43]. In total, we collected 13,828 reports from the Snyk database. Detailed information about each platform’s data from both data sources is presented in Table 2.

### 4.2. Package and Repository Information

**Longitudinal Data from Package Managers.** A key aspect of our study is the availability of historical data for all package managers. Libraries.io [37] is a free platform that aggregates publicly available open-source package information from package manager mirrors and registries on the internet. For all major platforms, including those analyzed in our study, Libraries.io provides dedicated profile pages that display the total number of packages available in each ecosystem’s registry. However, Libraries.io does not offer historical data, it only provides the current number of packages. To address this limitation, we used the Wayback Machine [44] to retrieve archived snapshots of Libraries.io’s website. From these archives, we collected historical data on the total number of packages for each package manager from 2017 to the present. We gathered two data points per year—one in January and one in July—enabling us to track the growth of each ecosystem over time.

**Metadata on Published Packages.** We collected the complete list of available versions for any of the reported packages from Libraries.io’s website. The data includes the version number and publication date of 11,316 unique packages. We will utilize this data to analyze the vulnerability lifespan based on the first vulnerable version and the first patched one.

### 4.3. Integrated Dataset Generation

**Unified Vulnerability Reports.** Although the OSV format has been introduced to streamline advisory databases, adoption remains inconsistent. Even among databases that use OSV, multiple formats exist. For example, affected versions of a CVE are sometimes presented as `events.introduced`, `events.fixed`, `events.last_affected`, and in other cases with arrays of versions. To address this, we adopted a unified format for affected versions, using brackets (`[]`) and parentheses (`()`). We established a unified data format to integrate the datasets from GitHub Advisory and Snyk. Each vulnerability report was standardized to include the following fields: `ADVISORY_ID`, `SOURCE`, `PUBLISH_DATE`, `CWES`, `CVES`, `ECOSYSTEM`, `PACKAGE_NAME`, `VERSION_RANGE`, `REFERENCES`.

**Removing Duplicate Reports** Detecting duplicates between advisories posed challenges due to inconsistent formatting, particularly for vulnerabilities without CVEs. We found 155 duplicate reports without a CVE with similar CWE, Package, Ecosystem, and Affected Versions. We did a manual verification on them to identify 7 duplicates. Overall, we removed 6,391 duplicate reports, ensuring a cleaner and more reliable dataset. Additionally, publish dates often differed as reports were published on different platforms at different times. When duplicates were identified, we prioritized Snyk reports over GitHub advisories because they typically contained more detailed information.

**Filtering Allowed CWEs.** We removed the reported vulnerabilities with CWEs that are discouraged or prohibited by MITRE. MITRE has flagged these CWEs as too broad or vague for actionable mitigation, discouraged from use. We found 6,770 vulnerability reports associated with these CWEs and removed them from our dataset to get a more accurate insights. Examples of such cases are CWE-400 (Uncontrolled Resource Consumption) and CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor). The list of discouraged and prohibited CWEs is provided in Appendix C.

### 4.4. Generated Dataset

The generated dataset includes three distinct and useful features:

- Information on total packages within each package manager and ecosystem.
- Unified vulnerability reports with a standardized format, excluding duplicates and including corrected affected versions.

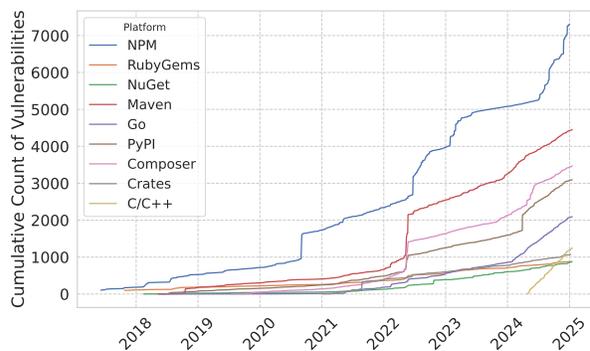


Figure 1. Number of vulnerabilities reported across ecosystems through the years. Reported vulnerabilities increase in every programming language, with an average of over 90 new vulnerabilities in each ecosystem per year.

- Package metadata including their vulnerability reports, affected versions, CVEs, and CWEs.

We generated a total of 31,267 vulnerability reports, 444 distinct CWEs, and 14,675 unique affected packages from 2017 to 2025. A summary of the collected data is presented in Table 2. We believe this dataset will significantly aid future researchers in studying open-source vulnerabilities and package-related information. We plan to keep the dataset updated with new vulnerability reports and package manager updates, providing the community with a unified and well-documented resource to support ongoing research.

## 5. Large-Scale Security Analysis in OSS

In this section, we aim to answer the research questions raised in Section 3. We analyze data from 10 programming languages: Node.js, PHP (Composer), Python, Java, Ruby, Go, Rust, .NET (C#), C, and C++. Eight corresponding package managers are also included in this evaluation: NPM, Packagist, PyPI, Maven, RubyGems, Go, Cargo, and NuGet.

### 5.1. Comparing Vulnerability vs Package Growth

**Increase in Total Vulnerabilities Across Languages.** Alongside the growing number of packages, we observe a significant rise in the total number of reported vulnerabilities across all platforms. As illustrated in Figure 1, the number of vulnerabilities has significantly increased in every programming language from 2017 to this date. We observe an average of over 90 new vulnerabilities reported in different ecosystems each year. This trend is particularly more significant in fast-growing platforms like NPM, Maven, and Composer, where the surge in both package creation and vulnerabilities has been substantial. In 2022, the number of reported vulnerabilities across different platforms significantly increased, with Maven and Composer reporting 2,497 and 1,592 vulnerabilities, respectively. We do not have sufficient data nor investigated deeply to make a scientific claim

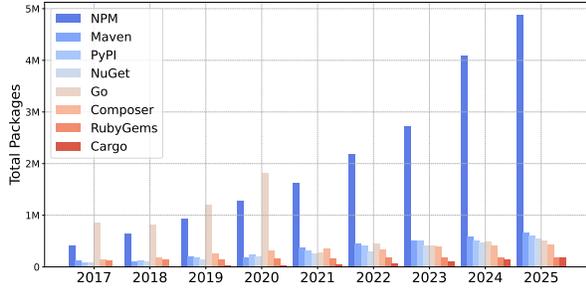


Figure 2. Overview of total packages existing in each package manager. Extracted from libraries.io.

about this surge. This increase can be attributed to several factors, including heightened government and regulatory awareness regarding cybersecurity [13] and high-impact supply chain attacks (such as Log4Shell [45], MoveIt [46], [47], and Solarwinds [48] Vulnerabilities)

**Increase in Total Number of Packages Across Ecosystems.** Across all studied platforms, we observe an average of 26.77% annual increase in the total number of packages. Figure 2 represents the total number of packages in each package manager from 2017 until January 2025. We observe the growing popularity and widespread adoption of open-source software, specifically with ecosystems such as NPM, PyPI, and Maven, since they have been experiencing exponential growth over time. There are exceptions where certain platforms show a temporary decline in package count during specific years, which are probably caused by the removal of outdated and unsupported packages from their registry. For instance, Golang experienced a notable package drop in 2021 following the release of Golang 1.16, which introduced major changes to its module count and dependency management system [49], [50], [51]. In particular, before Go Modules, Go used a global workspace, defined by the GOPATH environment variable, to manage dependencies where all projects shared the same workspace, making it difficult to manage multiple versions of the same dependency. This led to version conflicts, dependency issues, and challenges in ensuring consistent builds across projects.

**Comparing Growth Rates: Reported Vulnerabilities vs. Total Packages.** We compare the evolution of vulnerabilities with the growth of package managers over time to examine whether open-source communities have become more secure by analyzing two key factors: the total number of reported vulnerabilities and the total number of packages. We measure and compare the growth rates of these two variables to determine whether security practices have caused the open-source software supply chain to become more secure or whether the popularity and expansion are making them a better target for attackers. We define the growth rate of vulnerabilities and packages as the year-over-year percentage

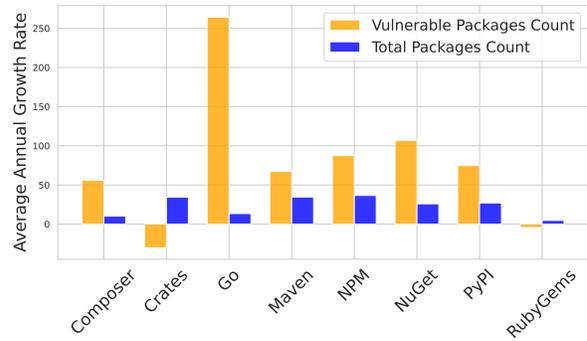


Figure 3. Average annual growth rate of vulnerable packages and total packages by platform. Vulnerabilities have grown at a much faster rate than total packages in nearly all ecosystems.

change in their respective counts. Figure 3 illustrates the comparative growth rates of vulnerabilities and packages across all measured ecosystems over the years.

Our analysis reveals that the growth rate of vulnerabilities has outpaced the growth rate of total packages across almost all ecosystems. We observe an average of 91% growth in the number of vulnerable packages compared to 25% growth in the total number of packages each year. This suggests that while the open-source software community continues to expand, the rate at which vulnerabilities are discovered is growing even faster, raising a major security concern. This can potentially be because of the higher community efforts in finding and reporting vulnerabilities. To investigate this further, later in Section 5.2, we analyze the decline in security trends from another point of view (i.e., vulnerability lifespan).

## 5.2. Vulnerability Lifespan and Concentration

We examine whether the growing popularity and community size of open-source software improve security or introduce new challenges. Additionally, we investigate whether vulnerabilities are concentrated in a few packages or spread across ecosystems. These are important questions to answer because one goal of this study is to compare trends across different ecosystems.

**Vulnerability Lifespan Across Ecosystems.** An important aspect of vulnerability analysis is to find out the time that has taken the package maintainers to resolve a reported vulnerability. We define time-to-fix as the duration between the disclosure of a vulnerability and the release date of a patched version. However, addressing this question is non-trivial, particularly due to inconsistencies in public data on different repositories as well as the lack of exact disclosure dates for many reported vulnerabilities. Moreover, in cases with a disclosure date, we found that the patched version was published before the listed date, making it difficult to provide an accurate analysis of the time-to-fix metric.

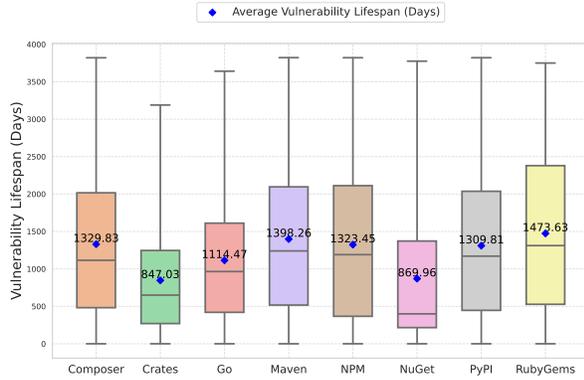


Figure 4. Vulnerability lifespans across ecosystems. Average vulnerability lifespan in different ecosystems varies from 847 to 1,473 days.

In our analysis, we focus on the vulnerability lifespan. We define the lifespan of a vulnerability as the time between the release date of the first vulnerable version of the related package and the release date of the first version in which the vulnerability is resolved. Figure 4 illustrates the output of lifespan analysis among different ecosystems. We do not present any results for C/C++ in this experiment because there are no unified package managers, which makes it impossible to specifically analyze the packages. We observed that the average vulnerability lifespan across different ecosystems varies from 847 days for Crates to 1,473 days for RubyGems. Comparing these numbers with the version update frequency of each ecosystem, we found out that the frequency of version updates of the vulnerable packages does not correlate with the previous insight. The data suggests that RubyGems and NPM have the highest number of published versions of different packages per day, with an average of 0.06 versions while we saw that it has the largest vulnerability lifespan. The definitions of time-to-fix and lifespan of vulnerabilities are presented in Section A of the Appendix.

**Vulnerability Lifespan Trends.** In Section 5.1, we examined the growth in the number of reported vulnerabilities across various ecosystems over the years. One possible explanation for this trend is the increased community efforts involved in vulnerability reporting. To investigate this, we conducted an experiment to assess whether the rise in contributor participation correlates with a reduction in the lifespan of vulnerabilities. However, our analysis reveals that greater contributor involvement does not necessarily lead to shorter lifespans. On the contrary, we observe that in most ecosystems, the average vulnerability lifespan has increased in recent years. As shown in Figure 5, the average lifespan of vulnerabilities across all platforms has grown from 1,056 days to 1,956 days—an increase of approximately 85%. This finding suggests that despite potentially higher levels of community engagement, vulnerabilities are persisting longer in the software supply chain.

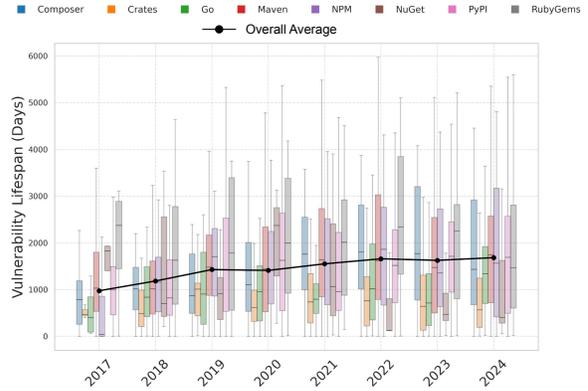


Figure 5. Vulnerability lifespans trend across platforms. While the community efforts have increased, we observe that the average lifespan has increased by 85% in the past years.

**Vulnerability Concentration Across Ecosystems.** We performed an analysis of the concentration of vulnerabilities in packages across ecosystems. Vulnerability concentration is calculated as the total number of reported vulnerabilities divided by the total number of vulnerable packages. A higher ratio of vulnerabilities per package indicates a focus on a smaller set of critical or frequently exploited packages, while a ratio closer to one implies a wider spread of vulnerabilities across many packages.

For example, in Composer, there are an average of 4.92 vulnerabilities per vulnerable package, signaling that certain packages are repeatedly targeted. Similar patterns are observed in C/C++ (3.96) and RubyGems (2.92). On the other hand, platforms like NPM, with an average of 1.75 vulnerabilities per package, demonstrate a broader distribution of vulnerabilities. Similarly, Go, Maven and PyPI show smaller ratios, suggesting a more even spread of vulnerabilities across packages. These findings, detailed in Figure 6, highlight the diversity of security challenges that ecosystems face. The presented results do not include intentionally malicious packages (Discussed in Section 5.4) since these packages are reported as a single vulnerability.

We also measured vulnerability concentration among all existing packages in an ecosystem, not only the vulnerable packages. We did not observe any noticeable changes in the ecosystem rankings based on vulnerability concentration compared to the previous measurement. Composer is still ranked with the highest overall ratio of 0.86 vulnerabilities per package, while NPM has the lowest at 0.15. Notably, NuGet and NPM both have fewer than 0.25 vulnerabilities per package, indicating significantly low concentration.

### 5.3. Vulnerability Distribution and Patterns Across Ecosystems.

In this section, we evaluate CWE trends across the studied platforms by analyzing the presence of reported

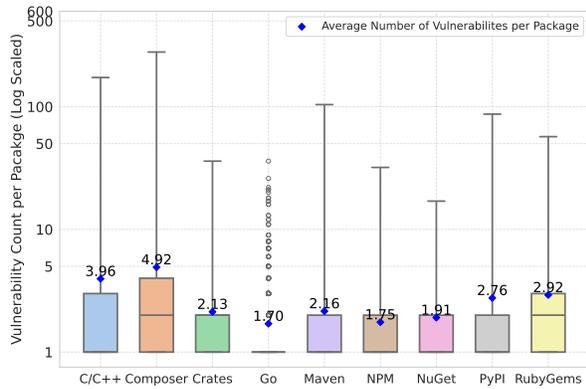


Figure 6. Vulnerability concentration in vulnerable packages across ecosystems. A higher concentration indicates a focus on a smaller set of critical or frequently exploited packages.

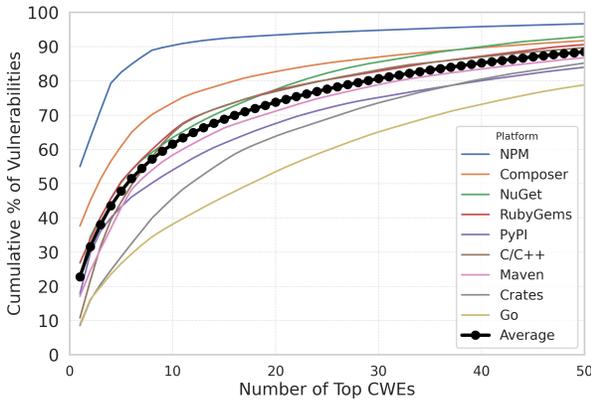


Figure 7. Cumulative graph of vulnerabilities based on top CWEs. A small proportion of CWEs lead to a large number of vulnerabilities. Seven CWEs account for over 50% of the vulnerabilities across the target platforms.

vulnerabilities in different ecosystems. As mentioned in Section 3, we aim to identify trends in vulnerability distribution for common CWEs and CWEs specific to ecosystems.

**Significance of Top CWEs.** We performed an analysis on vulnerability patterns across different ecosystems by investigating the most prevalent vulnerability categories. To this end, we generated a cumulative graph that relates the number of vulnerabilities to the number of top CWEs. This visualization helps assess whether vulnerabilities in a given platform are concentrated in a small set of CWE types or spread across a broader range. Figure 7 presents the cumulative percentage of vulnerabilities accounted for by the top 50 CWEs in each ecosystem. The results reveal a skewed distribution: in ecosystems such as NPM and Composer, a small number of CWE types contribute to a large proportion of vulnerabilities. In contrast, platforms like Go and Crates exhibit flatter curves, indicating a wider distribution of vulnerability types with less concentration in

CWE-506	0	0	11	0	1	3893	29	518	4
CWE-79	18	1208	20	141	728	589	97	344	214
CWE-22	15	130	34	175	259	571	39	187	45
CWE-94	39	168	4	27	133	151	181	114	50
CWE-352	3	203	0	32	318	42	8	59	39
CWE-1321	0	7	0	0	40	561	5	3	0
CWE-502	0	90	9	2	258	12	24	93	23
CWE-89	2	237	2	62	68	44	8	60	27
CWE-78	13	31	2	38	38	226	0	51	22
CWE-611	1	52	1	0	268	3	18	32	5
CWE-601	2	78	4	52	64	32	13	83	13
CWE-77	4	28	17	28	26	166	4	43	18
CWE-1333	2	7	8	3	43	139	6	51	55
CWE-918	4	56	0	46	75	54	11	53	10
CWE-862	0	29	0	30	225	6	2	14	2
CWE-416	135	0	78	3	9	14	42	10	8
CWE-863	1	54	7	72	103	7	11	37	1
CWE-770	9	17	27	71	69	5	14	46	7
CWE-125	126	1	36	16	13	22	1	36	4
CWE-434	1	139	0	9	30	23	9	24	3
CWE-362	21	4	67	33	16	4	18	57	8
CWE-787	96	0	37	10	25	2	17	22	4
CWE-122	135	0	12	0	3	1	47	9	4
CWE-295	9	8	5	29	83	8	10	35	21
CWE-532	2	11	6	54	40	13	7	34	2
CWE-522	0	3	2	22	110	5	2	14	0
CWE-120	66	0	35	8	6	6	0	30	3
CWE-347	6	11	19	24	28	29	3	25	7
CWE-835	10	7	9	28	40	6	4	30	5
CWE-476	53	0	26	21	3	2	3	16	5

Figure 8. Heatmap of the number of vulnerabilities from top 30 CWEs across different platforms. This suggests that a set of CWEs are common among different programming languages, while others are mainly observed in specific ecosystems.

specific CWEs. These findings highlight the need for ecosystem-specific mitigation strategies. For platforms where vulnerabilities are dominated by a few CWE types, targeted defenses focusing on those CWEs can be particularly effective. On the other hand, ecosystems with a more even distribution of CWEs may require broader, more comprehensive security measures. In the following, we analyze the most frequently observed CWEs and explore their characteristics to gain deeper insights into the nature of vulnerabilities.

**Popular CWE Types in Different Ecosystems.** We analyze the distribution of vulnerabilities across programming languages by identifying both common and ecosystem-specific CWEs. Figure 7 suggests that, on average, only seven CWEs in different ecosystems account for over 50% of the vulnerabilities. After merging the top CWEs from different platforms, we created a list of the top 28 different CWEs. Figure 8 illustrates a heatmap based on the number of vulnerabilities from a certain CWE in different platforms. This allows us to investigate security trends while highlighting the unique characteristics of certain ecosystems. In the following, we discuss the insights captured from this analysis.

**General Vulnerability Patterns.** We observe that multiple CWEs, including CWE-79 (Cross-Site Scripting), CWE-22 (Path Traversal), and CWE-94 (Code Injection) are common in several languages over the years and keep growing at a high rate. These common vulnerabilities emphasize fundamental challenges, specifically in web application security and data handling. Most web development environments face similar challenges,

including user-generated content, improper sanitization, and incorrect output encoding. Table 5 in Appendix provides more information for these common CWEs.

*Ecosystem-Specific Vulnerability Patterns.* Our analysis highlights a set of CWEs that are present in specific ecosystems. For instance, we observe that CWE-1321 (Prototype Pollution) is seen in 560 reports in NPM. This is caused by JavaScript’s prototype inheritance model, where objects can inherit properties from other objects (prototypes). Other examples include CWE-416 (Use After Free) and CWE-362 (Race Condition) which are mainly observed in C/C++ and Crates ecosystems due to their memory handling complexities. Table 5 in the Appendix provides examples of specific CWEs in different ecosystems and a brief description for each.

Overall, we observe that specific vulnerability patterns, such as XSS, Path Traversal, and Code Injection are present in multiple ecosystems. However, other classes of vulnerabilities, such as Prototype Pollution or Buffer Overflows, are deeply tied to language characteristics. Each of these vulnerability types potentially requires a different mitigation approach to defend against. Ecosystem-specific trends emphasize the need for custom analysis rules, threat modeling, and education tailored to the unique risks of each development environment. Figure 9 in the Appendix provides the trend of top CWEs in each ecosystem.

#### 5.4. Analysis of Packages with Malicious Intent (CWE-506)

CWE-506 (Embedded Malicious Code) represents an escalating threat in open-source ecosystems. Unlike other vulnerabilities, which often stem from negligence or poor design, these incidents are rooted in intentional abuse. That is, these vulnerabilities emerge from packages deliberately crafted to be exploited. The analysis of the dataset from 2017 to 2025 shows 4,456 reports classified under CWE-506. Notably, these incidents surged from just 38 reports in 2018 to a staggering 2,168 in 2024. This section explores the distribution of these malicious packages, their targeting strategies, and evolving attack vectors. Given their deliberate nature and operational difference, we treat CWE-506 incidents as a separate category, distinct from the other vulnerability trends discussed in Section 5.2.

**Distribution and Prevalence Across Ecosystems.** Table 3 presents the distribution of CWE-506 vulnerabilities across different ecosystems. It highlights the absolute count of malicious package reports, each ecosystem’s share of the total, and the proportion of its vulnerabilities stemming from intentional attacks.

Our analysis reveals a high concentration of malicious packages within the NPM and PyPI ecosystems, which together account for 99% of all CWE-506 reports. The situation is especially concerning in NPM, where nearly half (48.58%) of all reported vulnerabilities stem from packages designed to inflict harm rather than from unintentional bugs.

TABLE 3. DISTRIBUTION OF CWE-506 VULNERABILITIES ACROSS ECOSYSTEMS. EMBEDDED MALICIOUS CODE IS MAINLY REPORTED IN NPM AND PYPI, WITH MUCH FEWER INSTANCES IN OTHER ECOSYSTEMS.

Ecosystem	CWE-506 Count	% of All CWE-506	% of Ecosystem’s CWEs
NPM	3,893	87.37%	48.58%
PyPI	518	11.62%	13.92%
NuGet	29	0.65%	2.33%
Crates	11	0.25%	0.86%
RubyGems	4	0.09%	0.36%
Maven	1	0.02%	0.02%
Others	0	0.00%	0.00%
<b>Total</b>	<b>4,456</b>	<b>100.00%</b>	-

Several ecosystem-specific factors contribute to this trend. As noted in prior research [21], NPM’s low publication barriers, automatic dependency installation, and vast package ecosystem make it particularly attractive for attackers. PyPI also suffers from structural weaknesses that facilitate large-scale abuse. Bagmar et al. [52] describe how PyPI’s packaging model allows malicious packages to propagate easily through dependencies. More recently, Zheng et al. [53] found that PyPI lacks robust behavioral vetting and monitoring, leaving the ecosystem vulnerable to complex supply chain attacks. These systemic flaws across both platforms significantly increase their attack surface.

**Characteristics of Malicious Packages.** We investigated all 4,456 malicious packages to understand the tactics attackers use to ensure successful deception and broad reach.

*Package Naming Strategies.* Our analysis of naming strategies showed that attackers frequently attempt to camouflage their packages by mimicking legitimate libraries or adopting naming conventions that obscure their intent. We found that 3,171 packages (71.2%) use long names with more than 10 characters, and 2,999 (67.3%) include dashes in the name, these tactics are likely meant to appear consistent with conventional naming practices. Interestingly, 525 packages (11.8%) use the scoped format (e.g., @org/package), while 117 packages (2.6%) adopt very short names (less than 5 characters), perhaps to look core or internal. Additionally, 1,155 packages (25.9%) use names that closely resemble those of widely adopted libraries, highlighting the significant role of typosquatting and impersonation in attacker strategies. Note that these categories are not mutually exclusive.

*Version Targeting Patterns.* We also examined versioning patterns and found that attackers typically favor broad compatibility to maximize exposure. Our investigations show that 1,753 packages (39.3%) target all versions using wildcards such as ‘\*’, aiming for maximum compatibility. Meanwhile, 2,681 packages (60.2%) target specific versions, indicating a more controlled or environment-specific deployment. Only 22 packages (0.5%) employ version ranges. This data sug-

gests that attackers typically avoid precise versioning strategies and instead opt for broad compatibility to maximize exposure to more users.

**Attack Vectors and Techniques.** We observed three dominant attack vectors in packages classified under CWE-506: typosquatting, dependency confusion, and installation hook exploitation.

*Typosquatting Attacks.* Typosquatting attacks exploit human error by publishing packages with names that closely resemble popular libraries [54]. For example, attackers deployed variants such as `lodahs` (instead of `lodash`) and `jquery.js` (instead of `jquery`). We observed a significant spike in typosquatting attacks in March 2024 flooded PyPI with nearly 200 such malicious packages, all released on the same day. These packages mimicked well-known libraries, including `pygaqme` (instead of `pygame`) and `tensoflow` (instead of `tensorflow`).

*Dependency Confusion Attacks.* Dependency confusion represents a sophisticated supply chain attack where adversaries publish malicious packages to public repositories using names that match private, internal packages used by target organizations [55], [56]. This attack exploits the package resolution mechanism, where the package manager may prioritize public repositories over private ones, or developers may inadvertently install public packages when intending to use internal ones. Our dataset contains 525 scoped packages using the `@org/package` format, with manual analysis through a sample among these packages, we were able to confirm that there existed various dependency confusion attacks with examples that include `@ibm-ptc/greet-me` (targeting IBM’s private package ecosystem) [57], `@swiggy-private/analytics` (targeting the food delivery company Swiggy’s internal analytics infrastructure) [58]. These attacks are particularly dangerous because they target enterprise environments where the compromise of internal tooling can lead to widespread organizational breaches.

*Installation Hook Exploitation.* Installation hook exploitation represents a direct and immediate attack vector in the NPM ecosystem, abusing legitimate package lifecycle scripts to execute malicious code during package installation. NPM’s `package.json` specification includes several lifecycle hooks such as `preinstall` and `postinstall`, which were designed to allow packages to perform necessary setup tasks like compiling native extensions or configuring environment variables [56]. However, these hooks provide attackers with a powerful mechanism to execute arbitrary code with the same privileges as the user installing the package, often without explicit user consent or awareness. Examples from our dataset include `postinstall-dummy` [59], which explicitly advertises its malicious installation script functionality. These attacks are dangerous because they execute immediately upon installation, before developers have an opportunity

to review the package contents.

In this section, we covered three more common categories of CWE-506 attacks observed in our dataset. However, CWE-506 covers a more diverse scenarios that may not necessarily fall into these categories, but they are still tagged as a CWE-506. Notably, our analysis reveals that even trusted, well-known packages can become vectors for CWE-506 when maintainers deliberately inject malicious code into specific versions. One such case is discussed in the following:

**node-ipc Protestware (2022).** In March 2022, the maintainer of the popular `node-ipc` module in NPM, which has more than 600K weekly downloads, introduced geolocation-based file corruption logic in versions 10.1.1 and 10.1.2, targeting users in Russia and Belarus as a political statement against the invasion of Ukraine. The package contained malicious code, that targeted users with IP located in Russia or Belarus, and overwrote their files with a heart emoji [60]. This incident, later removed in version 10.1.3, marked one of the most prominent examples of “protestware”. It illustrates that even trusted packages can become vectors for CWE-506 when maintainers deliberately inject malicious code. This issue was assigned CVE-2022-23812.

Unlike other vulnerabilities, which often stem from negligence or poor design, these incidents are rooted in intentional abuse. The sheer volume, concentration, and strategic sophistication of malicious packages in NPM and PyPI demand dedicated detection mechanisms, supply chain hardening, and ecosystem-wide monitoring.

## 6. Discussion

**The Role of Package Managers.** Our findings reveal that Go’s vulnerability growth has been the highest among the studied ecosystems. However, Go has also made significant strides in security, such as with Go Modules, which replaced GOPATH to improve dependency management by removing deprecated and less-maintained packages. In contrast, Rust’s Crates.io is the only ecosystem with negative vulnerability growth. This may reflect the security nature of Rust or its lower popularity compared to more commonly targeted ecosystems. We further observe that the Go community has actively addressed common issues such as CWE-20 (Improper Input Validation) and CWE-22 (Path Traversal), both of which were among the top five vulnerability types in their ecosystem. However, recent community efforts have led to effective solutions, as seen in the growing family of safe Go libraries [61]. These libraries, such as `SafeText`, `SafeOpen`, and `SafeArchive` address common security issues in Go, including input validation and path traversal. This suggests that while vulnerabilities in Go have risen, the community has become more proactive, constantly developing defenses and keeping the ecosystem lively. That said, several strategies can be further improved to enhance the efficacy of package managers. For instance,

establishing clearer guidelines for package deprecation, promoting secure default configurations, and sharing more accurate guidelines for package submission can potentially assist developers in maintaining packages more securely. Overall, package managers play a critical role in reducing risks by encouraging secure defaults and actively maintaining packages.

**Limited Usage of Existing Protection Protocols.** Despite ongoing efforts to strengthen the security of open-source software through mechanisms such as automated vulnerability scanning, secure dependency management tools, and transparent builds such as Provenance [62] in the NPM registry, we observe that these protections remain largely underutilized. Although provenance was introduced over a year ago, it has not gained widespread adoption, even among high-risk packages like `rc`[63], `COA`[64], and `ua-parser` [65]. Notably, these same packages were referenced in Provenance’s launch materials, yet they still do not enable the feature. To better understand the adoption landscape, we examined the NPM’s monthly top 100 most-downloaded packages from `npmleaderboard.org` and found that only 12% had enabled Provenance. We observe that widely used packages such as `express` have not adopted the security mechanism. This gap between the availability of protection tools and their actual use leaves many repositories exposed to preventable attacks. The findings emphasize the need for stronger incentives, improved defaults, or tighter integration of these tools into standard development workflows to promote adoption.

## 7. Conclusion

In this paper, we provide a longitudinal analysis of security challenges in 10 different programming languages, revealing an alarming 98% annual increase in reported vulnerabilities—nearly four times the 25% growth rate of OSS packages. We also reveal the increase in lifespan of vulnerabilities, indicating a potential decline in the security of the ecosystems. Additionally, our findings highlight that a small number of Common Weakness Enumerations (CWEs) are responsible for the majority of reported vulnerabilities, suggesting that targeted mitigation could significantly improve overall OSS security. Furthermore, we discuss the alarming presence of intentionally malicious packages – particularly in the NPM and PyPI ecosystems – by analyzing their characteristics and attack vectors. These insights highlight an urgent need to rethink how we govern and secure open-source ecosystems – calling for stronger review processes, more effective automated detection tools, greater transparency in package publishing, and a more proactive, community-driven approach to managing software supply chain threats.

## References

- [1] J. P. Dantas, S. R. Silva, V. C. Gomes, A. N. Costa, A. R. Samersla, D. Geraldo, M. R. Maximo, and T. Yoneyama, “As-apy: A python library for aerospace simulation analysis,” in *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 15–24, 2024.
- [2] M. Ghiasi, T. Niknam, Z. Wang, M. Mehrandehz, M. Dehghani, and N. Ghadimi, “A comprehensive review of cyber-attacks and defense mechanisms for improving security in smart grid energy systems: Past, present and future,” *Electric Power Systems Research*, vol. 215, p. 108975, 2023.
- [3] A. N. Costa, F. L. Medeiros, J. P. Dantas, D. Geraldo, and N. Y. Soma, “Formation control method based on artificial potential fields for aircraft flight simulation,” *Simulation*, vol. 98, no. 7, pp. 575–595, 2022.
- [4] N. L. Wright, F. Nagle, and S. Greenstein, “Open source software and global entrepreneurship,” *Research Policy*, vol. 52, no. 9, p. 104846, 2023.
- [5] H. Yang, X.-Y. Liu, and C. D. Wang, “Fingpt: Open-source financial large language models,” *arXiv preprint arXiv:2306.06031*, 2023.
- [6] J. M. Tyler, B. J. Murch, C. Vasilakis, and R. M. Wood, “Improving uptake of simulation in healthcare: User-driven development of an open-source tool for modelling patient flow,” *Journal of Simulation*, vol. 17, no. 6, pp. 765–782, 2023.
- [7] vUpgradeU, “Open source development: A double-edged sword,” 2023. Accessed: 2024-10-15.
- [8] CyberSaint, “Impact of using open source software on cybersecurity,” 2023. Accessed: 2024-10-15.
- [9] The Hacker News, “27 malicious pypi packages with info-stealing capabilities discovered,” 2023. Accessed: 2024-10-14.
- [10] Sysdig, “The hidden economy of open source software,” 2023. Accessed: 2024-10-15.
- [11] Phylum, “Dozens of npm packages caught attempting to deploy reverse shell,” 2023.
- [12] Bleeping Computer, “Hundreds of malicious python packages found stealing sensitive data,” 2023. Accessed: 2024-10-14.
- [13] United States Congress, “Securing open source software act.” <https://www.govinfo.gov/content/pkg/BILLS-117s4913is/pdf/BILLS-117s4913is.pdf>, 2021. Accessed: 2024-10-07.
- [14] The White House, “Summary of the 2023 request for information on open source software security.” <https://www.whitehouse.gov/wp-content/uploads/2024/08/Summary-of-the-2023-Request-for-Information-on-Open-Source-Software-Security.pdf>, 2024. Accessed: 2024-10-07.
- [15] Cybersecurity and I. S. A. (CISA), “Cisa open source software security roadmap.” <https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap>, 2023. Accessed: 2025-01-22.
- [16] GeeksforGeeks, “Top web development languages.” Accessed: 2024-12-16.
- [17] LambdaTest, “Best languages for web development.” Accessed: 2024-12-16.
- [18] BrowserStack, “Best languages for web development.” Accessed: 2024-12-16.
- [19] S. Academy, “Best languages for web development.” Accessed: 2024-12-16.
- [20] M. Alfadel, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in python packages,” *Empirical Software Engineering*, vol. 28, no. 59, 2023.
- [21] M. Zimmermann *et al.*, “Small world with high risks: A study of security threats in the npm ecosystem,” in *28th USENIX Security Symposium (USENIX Security 19)*, pp. 995–1010, 2019.

- [22] R. Duan *et al.*, “Towards measuring supply chain attacks on package managers for interpreted languages,” *arXiv preprint arXiv:2002.01139*, 2021. Presented at NDSS.
- [23] N. Zahan *et al.*, “Do software security practices yield fewer vulnerabilities?,” in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 231–240, IEEE, 2023.
- [24] A. Decan, T. Mens, and M. Claes, “An empirical comparison of dependency issues in oss packaging ecosystems,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 2–12, IEEE, 2017.
- [25] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.
- [26] R. Kikas *et al.*, “Structure and evolution of package dependency networks,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 102–112, IEEE, 2017.
- [27] A. Decan, T. Mens, and E. Constantinou, “On the impact of security vulnerabilities in the npm package dependency network,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 181–191, 2018.
- [28] A. Zerouali *et al.*, “On the impact of security vulnerabilities in the npm and rubygems dependency networks,” *Empirical Software Engineering*, vol. 27, no. 5, p. 107, 2022.
- [29] I. Pashchenko *et al.*, “Vulnerable open source dependencies: Counting those that matter,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018.
- [30] MITRE, “Common vulnerabilities and exposures (cve).” <https://cve.mitre.org/about/>, 2024. Accessed: 2024-10-13.
- [31] MITRE, “Mitre corporation.” <https://www.mitre.org/>. Accessed: 2024-10-13.
- [32] C. Team, “Common weakness enumeration (cwe).” <https://cwe.mitre.org/>, 2024. Accessed: 2024-10-13.
- [33] N. I. of Standards and T. (NIST), “National vulnerability database (nvd).” <https://nvd.nist.gov/>. Accessed: 2024-10-13.
- [34] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Madhila, and L. Williams, “What are weak links in the npm supply chain?,” in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 331–340, 2022.
- [35] “github/advisory-database,” Oct. 2024. [Online; accessed 3. Oct. 2024].
- [36] Snyk, “Snyk vulnerability database.” <https://snyk.io/vuln>. Accessed: 2024-10-13.
- [37] Libraries.io, “Libraries.io - discover open source libraries.” <https://libraries.io/>. Accessed: 2024-10-13.
- [38] Github.com, “Build software better, together.” <https://github.com/>. Accessed: 2024-10-13.
- [39] “Open Source Vulnerability format - Open Source Vulnerability schema,” Oct. 2024. [Online; accessed 3. Oct. 2024].
- [40] M. Kluban, M. Mannan, and A. Youssef, “On measuring vulnerable javascript functions in the wild,” in *Proceedings of the 2022 ACM on Asia conference on computer and communications security*, pp. 917–930, 2022.
- [41] D. Sushma, M. Nalini, R. A. Kumar, and M. Nidugala, “To detect and mitigate the risk in continuous integration and continous deployments (ci/cd) pipelines in supply chain using snyk tool,” in *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pp. 1–10, IEEE, 2023.
- [42] T. Kim, S. Park, and H. Kim, “Why johnny can’t use secure docker images: Investigating the usability challenges in using docker image vulnerability scanners through heuristic evaluation,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 669–685, 2023.
- [43] D. Noever, “Can large language models find and fix vulnerable software?,” *arXiv preprint arXiv:2308.10345*, 2023.
- [44] Internet Archive, “Wayback machine,” 2024. Accessed: 2024-10-14.
- [45] IBM, “Log4shell: The vulnerability explained,” 2024. Accessed: 2024-10-14.
- [46] Lawrence Abrams, “US govt offers \$10 million bounty for info on Clop ransomware.” <https://www.bleepingcomputer.com/news/security/us-govt-offers-10-million-bounty-for-info-on-clop-ransomware/>, Accessed: 06-30-2023.
- [47] Cisco Talos, “Active exploitation of the MOVEit Transfer vulnerability by Clop ransomware group.” <https://blog.talosintelligence.com/active-exploitation-of-moveit/>, Accessed: 06-30-2023.
- [48] Center of Internet Security, “The SolarWinds Cyber-Attack: What You Need to Know.” <https://www.cisecurity.org/solarwinds>, Accessed: 06-30-2023.
- [49] G. Team, “Go modules reference.” <https://go.dev/ref/mod>, 2023. Online; Accessed: 2024-10-12.
- [50] G. Team, “Go proxy.” <https://proxy.golang.org/>, 2023. Online; Accessed: 2024-10-12.
- [51] M. Valais, “Using the GO111MODULE everywhere!” <https://maelvls.dev/go111module-everywhere/#go111module-with-go-116>, 2021. Online; Accessed: 2024-10-12.
- [52] A. Bagmar, J. Wedgwood, D. Levin, and J. Purtilo, “I know what you imported last summer: A study of security threats in thepython ecosystem,” 2021.
- [53] W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu, “An empirical study of malicious code in pypi ecosystem,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 166–177, 2023.
- [54] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pp. 23–43, Springer, 2020.
- [55] A. Birsan, “Dependency confusion: How i hacked into apple, microsoft and dozens of other companies.” <https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610>, 2021. Accessed: 2025-05-15.
- [56] Snyk, “npm dependency confusion attack targeting gxm reference.” <https://snyk.io/blog/npm-dependency-confusion-attack-gxm-reference/>, 2022. Accessed: 2025-05-15.
- [57] Snyk, “SNYK-JS-IBMPTCGREETME-8518693: Malicious Package in @ibm-ptc/greet-me.” <https://security.snyk.io/vuln/SNYK-JS-IBMPTCGREETME-8518693>, 2024. Accessed: 2025-05-25.
- [58] Snyk, “SNYK-JS-SWIGGYPRIVATEANALYTICS-8553970: Malicious Package in @swiggy-private/analytics.” <https://security.snyk.io/vuln/SNYK-JS-SWIGGYPRIVATEANALYTICS-8553970>, 2024. Accessed: 2025-05-25.
- [59] Snyk, “SNYK-JS-POSTINSTALLDUMMY-2847431: Malicious Package in postinstall-dummy.” <https://security.snyk.io/vuln/SNYK-JS-POSTINSTALLDUMMY-2847431>, 2022. Accessed: 2025-05-25.

[60] Snyk, “SNYK-JS-NODEIPC-2426370: Malicious Package in node-ipc.” <https://security.snyk.io/vuln/SNYK-JS-NODEIPC-2426370>, 2022. Accessed: 2025-05-16.

[61] G. B. Hunters, “The family of safe golang libraries is growing,” 2023. Accessed: 2024-10-14.

[62] B. D. Harrison, Philip, “Introducing npm package provenance,” Apr. 2023. Accessed: 2024-10-14.

[63] npm, “rc - npm package,” 2024. Accessed: 2024-10-14.

[64] npm, “coa - npm package,” 2024. Accessed: 2024-10-14.

[65] npm, “ua-parser-js - npm package,” 2024. Accessed: 2024-10-14.

[66] “CWE - CWE-79: Improper Neutralization of Input During Web Page Generation (‘Cross Site Scripting’),” 2024. Accessed: 2024-10-14.

[67] “CWE - CWE-22: Improper limitation of a pathname to a restricted directory (‘path traversal’),” 2024. Accessed: 2024-10-14.

[68] “CWE - CWE-94: Improper control of generation of code (‘code injection’),” 2024. Accessed: 2024-10-14.

[69] “CWE - CWE-1321: Improper Handling of Prototype Pollution,” 2024. Accessed: 2024-10-14.

[70] “CWE - CWE-122: Heap-based Buffer Overflow,” 2024. Accessed: 2024-10-14.

[71] “CWE - CWE-416: Use After Free,” 2024. Accessed: 2024-10-14.

[72] “CWE - CWE-787: Out-of-Bounds Write,” 2024. Accessed: 2024-10-14.

[73] Microsoft, “Range operator.” Microsoft Docs, 2024. Accessed: 2024-10-14.

[74] “CWE - CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (‘Race Condition’),” 2024. Accessed: 2024-10-14.

## Appendix

### 1. Terminology Definitions.

Below, we provide the formulas and definitions for each term used in our study to ensure clarity and precision.

**Vulnerability Lifespan** The vulnerability lifespan measures the duration a vulnerability persists in a package before being patched. Since a single package may have multiple affected version ranges, we calculate the total lifespan by summing the durations of all such ranges. This provides a comprehensive view of how long vulnerabilities remain unresolved across different versions.

$$\sum_{i=1}^n (\text{First Patched Version Release Date}_i - \text{First Affected Version Release Date}_i)$$

Here,  $n$  represents the number of affected version ranges for a given package.

**Time-to-Fix** Time-to-Fix measures the duration between the disclosure of a vulnerability and the release of a patched version of the affected package. It is calculated by subtracting the disclosure date of a report from the release date of the first unaffected version. This metric highlights how quickly ecosystems respond to disclosed vulnerabilities.

## 2. Rise and Fall of Top CWEs

Figure 9 illustrates different ecosystem’s top five most reported CWEs over time. These CWEs have either become more prominent or less frequent due to improvements in ecosystem security practices.

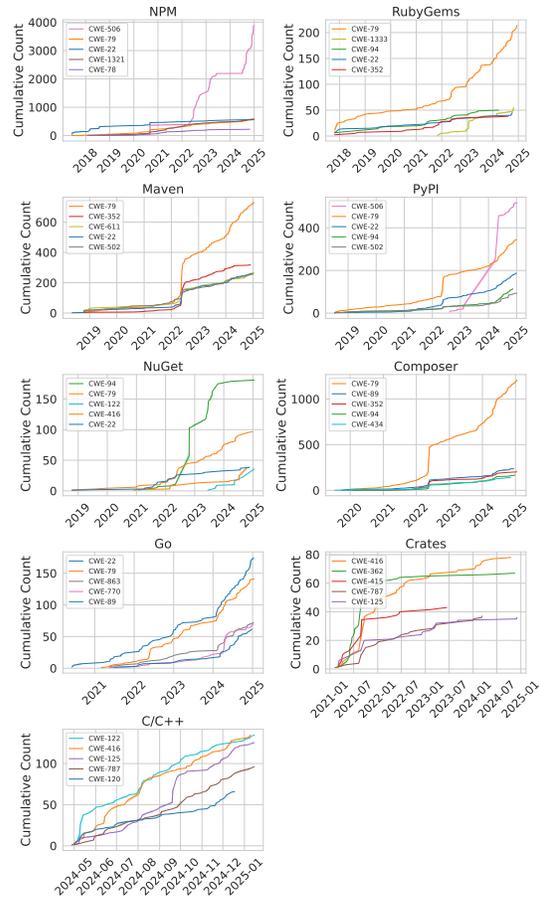


Figure 9. Top five most reported CWEs in each ecosystem over time.

### 3. Discouraged and Prohibited CWEs

Table 4 includes the CWEs that are marked as discouraged or prohibited to use by MITRE.

### 4. Common and Ecosystem-Specific CWEs

Table 5 includes examples for common and ecosystem-specific CWEs.

TABLE 4. DISCOURAGED AND PROHIBITED COMMON WEAKNESS ENUMERATION (CWE) AND THEIR DESCRIPTION

CWE ID	Description	CWE ID	Description
CWE-19	Data Processing Errors	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor
CWE-20	Improper Input Validation	CWE-255	Credentials Management Errors
CWE-74	Improper Neutralization of Special Elements in Output	CWE-264	Permissions, Privileges, and Access Controls
CWE-75	Failure to Sanitize Special Elements into a Different Plane	CWE-265	Privilege / Sandbox Issues
CWE-118	Incorrect Access of Indexable Resource ('Range Error')	CWE-269	Improper Privilege Management
CWE-119	Improper Restriction of Operations	CWE-274	Improper Handling of Insufficient Privileges
CWE-138	Improper Neutralization of Special Elements	CWE-275	Permission Issues
CWE-284	Improper Access Control	CWE-285	Improper Authorization
CWE-287	Improper Authentication	CWE-300	Channel Accessible by Non-Endpoint
CWE-310	Cryptographic Issues	CWE-311	Missing Encryption of Sensitive Data
CWE-330	Use of Insufficiently Random Values	CWE-345	Insufficient Verification of Data Authenticity
CWE-372	Incomplete Internal State Distinction	CWE-391	Unchecked Error Condition
CWE-400	Uncontrolled Resource Consumption	CWE-435	Improper Interaction Between Multiple Entities
CWE-438	Behavioral Problems	CWE-610	Externally Controlled Reference to a Resource in Another Sphere
CWE-657	Violation of Secure Design Principles	CWE-662	Improper Synchronization
CWE-664	Improper Control of a Resource Through its Lifetime	CWE-665	Improper Initialization
CWE-668	Exposure of Resource to Wrong Sphere	CWE-680	Integer Overflow to Buffer Overflow
CWE-682	Incorrect Calculation	CWE-690	Unchecked Return Value to NULL Pointer Dereference
CWE-692	Incomplete Cleanup	CWE-693	Protection Mechanism Failure
CWE-697	Incorrect Comparison	CWE-703	Improper Check or Handling of Exceptional Conditions
CWE-707	Improper Neutralization	CWE-755	Improper Handling of Exceptional Conditions
CWE-788	Access of Memory Location After End of Buffer	CWE-834	Excessive Iteration
CWE-840	Business Logic Errors	CWE-1047	Modules with Circular Dependencies
CWE-1056	Invocation of a Control Element at a Deep Layer	CWE-1068	Inconsistency Between Implementation and Documented Design
CWE-1103	Use of Platform-Dependent Third Party Components	CWE-1118	Insufficient Documentation of Error Handling Techniques
CWE-1119	Excessive Use of Unconditional Branching	CWE-1125	Excessive Attack Surface

TABLE 5. COMMON VULNERABILITY PATTERNS AND ECOSYSTEM-SPECIFIC CWES ACROSS SOFTWARE PLATFORMS.

CWE	Ecosystem(s)	Description
CWE-79 (Cross-Site Scripting) [66]	Composer, Maven NPM, PyPI	XSS remains a major security concern in web applications, especially dominating Composer (1208), Maven (728), NPM (589), and PyPI (344).
CWE-22 (Path Traversal) [67]	NPM, Maven PyPI	Path traversal enables attackers to access files outside limited boundaries. NPM (571), Maven (259), and PyPI (187) show the highest number of reports from this category.
CWE-94 (Code Injection) [68]	NuGet, Composer NPM, Maven, PyPI	Code injection is reported with high frequency in NuGet (181), Composer (168), NPM (151), and Maven (133), followed by PyPI (114). This indicates a high risk in dynamic server-side applications, especially those that handle user input.
CWE-1321 (Prototype Pollution) [69]	NPM	Over 560 reports of prototype pollution in NPM highlight the necessity for improved validation mechanisms in libraries that perform object handling.
CWE-122 (Heap Buffer Overflow) [70]	C/C++	The top reported CWE for C/C++ is heap-based buffer overflow (135 reports). This reflects the challenges of manual memory management, leading to memory corruption issues.
CWE-416 (Use After Free) [71]	C/C++, Crates	Memory usage after it has been freed leads to undefined behavior or security vulnerabilities. This is mostly observed in C (135) and Crates (78).
CWE-787 (Out-of-Bounds Write) [72]	NuGet	This is mostly seen in packages where memory safety practices are crucial. The trends show that this CWE type has decreased in NuGet, likely due to better memory safety practices such as the introduction of Index and Range types in C# 8 [73].
CWE-362 (Race Condition) [74]	C/C++, Crates	This has been a prominent vulnerability in C and Crates in the past years. However, the reports on this CWE have become less frequent, reflecting improved concurrency handling.