Hurry Up and Wait: Malware Detection Timelines and Minimum Release Age for npm

Dominic Tassio
University of Kansas
Lawrence, Kansas, USA
dominic.tassio@ku.edu

Thanoshan Vijayanandan
University of Calgary
Calgary, CA
thanoshan.vijayanand@ucalgary.ca

Caleb Morse
University of Kansas
Lawrence, Kansas, USA
caleb.morse@ku.edu

Lorenzo De Carli
University of Calgary
Calgary, CA
lorenzo.decarli@ucalgary.ca

Drew Davidson
University of Kansas
Lawrence, KS, USA
drewdavidson@ku.edu

Abstract

Setting a minimum release age is a new feature of npm, pnpm, Yarn, and other package managers, introduced to provide a mechanism for mitigating the impact of malware being uploaded to npm. However, no evidence-based analysis has been performed to determine how this security mechanism should be tuned to make effective use of it. To close this gap, we analyze the GitHub Advisory Database to collect data on the timeline of malware detection. Building upon prior work on measuring malicious package detection and popularity-weighted impact, we combine historical malicious package detection and npm download metrics to estimate minimum release age thresholds that can provide developers with confidence that updated releases contain no malicious code. To our knowledge, we are the first to provide evidence based recommendations for setting the recently introduced minimum release age feature. By conducting this study, we hope to provide real world benefits to developer security. Based on our study of malware detection time using the GitHub Advisory Database, we find that a general recommendation of setting the minimum release age to 8 days provides significant security benefits with little to no negative impact on developers.

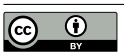
CCS Concepts

• **Security and privacy** → **Malware and its mitigation**; *Vulnerability management*; • **Software and its engineering** → *Software libraries and repositories.*

Keywords

minimum release age, software supply chain security, package ecosystems, malicious packages, npm

ACM Reference Format:
Dominic Tassio, Thanoshan Vijayanandan, Caleb Morse, Lorenzo De Carli, and Drew Davidson. 2026. Hurry Up and Wait: Malware Detection Timelines and Minimum Release Age for npm. In *Conference on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '26), October 06, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3848003.3848013



SCORED '26, Prague, Czech Republic

ACM ISBN 979-8-4007-3009-2/2026/10
https://doi.org/10.1145/3848003.3848013

1 Introduction

Package ecosystems underlie many of the most popular programming languages. These package ecosystems allow developers to quickly and automatically integrate reusable code, authored by a variety of package providers with diverse update strategies. The automated nature of package management allows developers to largely operate without knowledge of when and how updates are being made to their package dependencies. The reliability, interoperability, and security surface of the packages upon which a developer depends can shift as changes are made to their code via updates. The open nature of package ecosystems allows package providers to freely contribute to the ecosystem. However, it is also a means by which malicious or vulnerable code can be introduced into the ecosystem, compromising codebases and other packages that incorporate an infected package. Indeed, in the last decade, several such supply-chain attacks have caused high-profile incidents and large-scale compromises [3–5, 9, 23, 28, 32, 36–38].

One way in which a developer can be exposed to such attacks is when a *malicious update* is made to a previously-benign package [11, 35]. This threat vector is well known to the software supply chain community. Developers may intentionally degrade functionality for personal reasons, a new member may be added to a package developer team who has ill intent, or a benign package developer may have their credentials compromised and used to insert malware. When a malicious update occurs, developers who use the infected package are exposed to the code through their package dependencies. This can happen to developers either when installing a new dependency for their project or when updating an already existing dependency.

Fortunately, the open nature of package ecosystems also serves as a security benefit. Security analysts and members of the ecosystem community can freely scrutinize code, including updates, for undesired behavior and report issues to the maintainers of the package registry. Detection of these malicious packages is a reactive process, and developers depend on the security apparatus of the ecosystem to do so. Once a package update is uploaded to the package ecosystem, it may be analyzed for malicious code and/or behavior; at the same time, early adopters and security organizations may begin using and scrutinizing the package. In the best case,

the malicious version is removed before a developer can download it. As the detection of malicious packages occurs after they have been uploaded to the npm registry, time is needed to allow for the detection and removal of the malicious packages.

Several package managers now provide a configuration option that allows a user to specify a minimum release age for packages. When set, the package manager excludes versions newer than the configured threshold from dependency resolution. This gives external detection and remediation processes time to act before the version becomes eligible for installation. However, given the recency of minimum release age configurations, developer awareness and best-practices surrounding this feature are likely lacking, making it a strong candidate for research.

The focus of our work is to determine how the minimum release age can be used to avoid malicious package updates. As the feature is configurable, the question arises as to what to set as the age. A tradeoff exists here: waiting longer can allow more scrutiny, thus decreasing the probability that a package may contain malicious code; at the same time, it may prevent the incorporation of new features and/or security updates that may be provided by the new version. To present an evidence based recommendation for this configuration option, we conduct a study of security advisories involving malicious code uploaded to npm.

The rest of our paper is structured as follows: Section 2 provides background on minimum release age policy. Section 3 gives an overview of how malicious updates can be avoided by enforcing delays. Section 4 describes how we collect historical incident reports to determine an optimal minimum release age. Section 5 provides our evaluation for deriving recommended minimum release age values. We summarize these results and discuss their implications in section 6, and describe future work beyond these results in section 7. We compare our work here with prior results in section 8 and conclude in section 9.

2 Background

In this section, we provide the necessary context on package ecosystems to understand the need for and implementation details of delay windows. We detail package ecosystems generally in section 2.1 and npm specifically in section 2.2. We describe malicious updates in section 2.3. Finally, we provide the technical background for setting the minimum release age in section 2.4.

2.1 Package Ecosystems

Many modern languages are supported by systems that provide and programmatically manage re-usable software components. We refer generically to the re-usable components as *packages*, and the systems and developers that manage them as *package ecosystems*. Different ecosystems use distinct names for each of their components. Figure 1 introduces generic terminology that captures the common details of package ecosystems of interest for our work.

User: The user of a package ecosystem may interact in a variety of ways with the rest of the ecosystem. Most obviously, a developer of an application may manually issue commands to download, configure, and incorporate code into their application, which we will term a first-party application. A user may also be using the package ecosystem indirectly, scripting the installation and configuration of packages as part of a larger system deployment.

In addition to the development of a first-party application, the user may be the developer of a package that itself depends on other packages. Thus, the user may not only be a consumer of package code but also a contributor to the registry.

Registry: We use the term *registry* to refer to the online repository of packages as well as the interface that services requests from the frontend. Packages are uploaded to the registry from a variety of developers with different security postures, release cycles, and priorities. Many ecosystems allow pseudonymous developer accounts. The maintainers of package registries ingest and store large amounts of code. While security teams take action to remove code that has been deemed malicious from the registry, they typically do not enforce code audits before a package is released.

Frontend: We use the term *frontend* to refer to the software tool that accepts user commands for registry interaction. The core functionality of the frontend is to retrieve, install, and configure a specified package. However, frontends may also have the ability to audit or analyze a package, either by reading metadata directly from the registry or by referencing third-party databases. These auxiliary checks help provide assurance regarding package installation, even when the use of the frontend is automated.

Requested package: In the most basic form of interaction with the ecosystem, the user commands the frontend to retrieve a named package from the registry. While most package ecosystems allow the user to specify a particular version of the package, in most cases, the default behavior is to retrieve the latest version.

Dependencies: A key feature of package registries is that individual packages may depend on one another. Thus, when any requested package is specified by a user, the frontend will read the package metadata to collect other packages that are necessary for the requested package to function. These dependencies will themselves be retrieved, configured, and installed, potentially inducing a collection of additional dependencies. Through this transitive process, a large collection of packages may be installed even when a single package is requested. This reality means that a user may be unaware of all the packages they are being exposed to on a single invocation of the frontend.

2.2 npm Ecosystem

For the purpose of presenting a running example, and because it is the largest ecosystem in terms of total packages and use, we focus on the npm ecosystem. At over 3 million registered packages, npm is the largest software package registry. Primarily for Node.js and JavaScript, npm serves a diverse set of packages with various scopes and functionalities.

Packages on npm follow a permissive structure that is identified by a `package.json` file. Importantly, and as required by npm, a package must define a name and version in a `package.json`. In addition to the name and version, a package can define many other standard and recognized properties.

A key feature of npm, enabled by the `package.json` file, is the ability to define dependencies on other packages by specifying a name and version range. The `"dependencies"` property is the standard way for packages on npm to depend on other packages and the

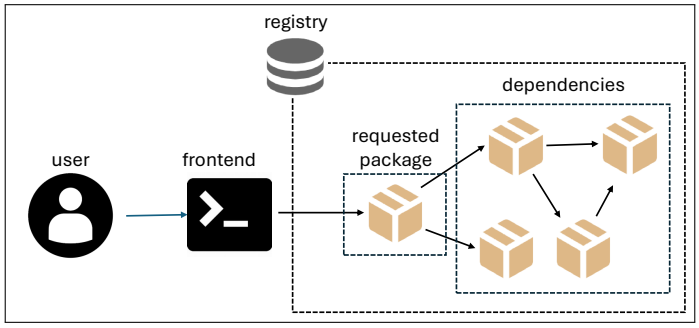


Figure 1: Generic Package Ecosystem Diagram

primary mechanism by which developers can specify dependencies for their projects.

2.3 Malicious Updates

A recurring problem for developers using npm packages is the possibility of a malicious update to a package they are currently using. This fear is well founded, as there have been numerous high profile compromises of packages within the npm ecosystem [3–5, 9, 23, 28, 32, 36–38]. Due to the increasing frequency of these malicious updates, npm and other package managers that use the npm registry have introduced various features to prevent or mitigate this issue.

A significant milestone for security within the npm ecosystem was the introduction of `package-lock.json` files. This made package installation a reproducible process, as it ensures that the versions of the packages installed are the exact versions stored in the lock file. Conversely, if dependencies are fetched purely based on the dependency list in `package.json`, the system will fetch the most recent version of each dependency package, potentially downloading and installing versions different from the ones that the original package maintainer downloaded and tested. However, even if the use of `package-lock.json` has removed some nondeterminism in the installation process, it cannot prevent every situation where a malicious package could be installed. Further, there are situations where a developer may intentionally choose to update a package or packages. It also does not help in the case of installing a new package.

2.4 Setting a Minimum Release Age

On 11 February 2026, npm released version v11.10.0 of the npm CLI, which included support for the `min-release-age` configuration option [30]. Setting this option to a number measured in days restricts npm to installing packages that have been available for the specified number of days. With the addition of this option, npm now joins other popular package managers, such as pnpm and Yarn, in supporting this feature.

The mechanisms that pnpm and Yarn use to support this feature differ in detail but not in function compared to npm. For pnpm, the setting is specified in `pnpm-workspace.yaml` as `minimumReleaseAge` and is given in minutes, defaulting to 1 day [34]. For Yarn, `npmMinimalAgeGate` can be set in `.yarnrc.yml` and is specified in minutes or in shorthand strings like `"1w"`.

All these package managers provide the ability to exclude certain packages from this setting. For npm, the relevant option is `min-release-age-exclude`; for pnpm, it is `minimumReleaseAgeExclude`; and for Yarn, it is `npmPreapprovedPackages`.

3 Overview

We examine the need for a minimum release age through the lens of a conceptual timeline depicted in fig. 2. This timeline describes the basic milestones in an incident involving a malicious package update, along with the possible remediation paths.

3.1 Malicious Update Incident Timeline

1a) Benign Upload: In this case, we assume that the initial upload of a package includes only benign functionality, although it may include latent vulnerabilities and/or lack functionality.
1b) Malicious Upload: In the other case, we assume that the initial upload of a package does contain malware. In our conceptual timeline, we then proceed to the security advisory step. Otherwise, we continue with the benign assumption.
2) Benign Updates: We assume that zero or more benign updates have been made to a package during the course of its development. These updates may include bug fixes, additional functionality, or refactoring changes.
3) Malicious Update: In order for an attack to qualify as a malicious update incident, there is necessarily a point at which an adversary introduces a version of an initially benign package with malicious functionality. For the purpose of our work, we do not focus on the specifics of what malicious functionality is introduced. The malicious update may be an alteration to the code that causes undesired functionality to be included in the operation of the codebase, or the attack might cause malicious functionality to be directed towards the installer of the package. Similarly, we do not focus on the means by which the malicious update has been achieved. For example, a member of the maintainer team might have their authentication keys compromised, allowing a malware author to upload their attack on behalf of the maintainer. A malicious developer might be able to use social engineering to be added to the maintainer team, or a previously-benign developer might become malicious and inject an attack.
4) Security Advisory: At some point, we assume that the malicious update will be discovered, either by victims of the attack reporting the incident or by security analysts uncovering the malicious

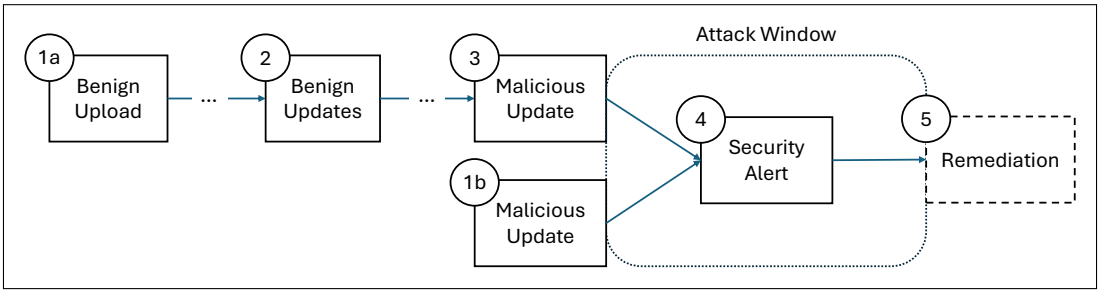


Figure 2: Conceptual Timeline for a Malicious Update Incident and Remediation

behavior. We note that the actual methods by which an attack is discovered are beyond the scope of this work. Similarly, our focus on using minimum release age means that we only measure those incidents for which the attack is eventually discovered. For a security advisory to exist in our framework, it must be uploaded to a vulnerability database that verifies and republishes the advisory. We consider a security advisory to be the point in time at which the danger of falling victim to the corresponding malicious update has passed. We distinguish first detection from advisory publication. As our dataset provides the publication timestamp of the advisory, our analysis uses the advisory publication as a proxy endpoint. This proxy does not necessarily coincide with the time at which the malicious version was detected or became unavailable.

Our study primarily focuses on npm and the GitHub Advisory Database. However, our findings equally apply to package ecosystems in which there is no such database. In those cases, the incident concludes only when a remediation update is provided (Step 5, described below).

5) Remediation Update: When a security advisory has been addressed, a new version of the package may be released that mitigates the malicious update. We note that this step may not occur for a variety of reasons: if the codebase is considered complete, the full mitigation may consist solely of rolling back the malicious update; the codebase may be abandoned by its maintainer(s), in which case an update may be necessary but will nevertheless not be provided; or an update may be planned but has not yet been released. We consider the rollback of a package version that simply removes the malicious code to be sufficient as remediation for the purpose of our study.

3.2 Attack Windows

The key focus of our study is to characterize how the user of a package ecosystem can avoid installing or updating a package during the period in which the malicious version is active. Should the package frontend consult with an advisory database, the user need only wait until Step 4 occurs. If the registration of an incident is not sufficient to prevent the malicious version from being installed, such as in a scenario where a registry cannot or will not act, the user must wait until Step 5. We refer to this period as the *attack window*.

Various factors might affect the length of the attack window, such as the availability of scanning tools, the stealthiness of the attack, and the amount of scrutiny being applied to packages. Thus, we set out to offer guidance on how long the user needs to wait in order to surpass the attack window.

3.3 Risk Profiles

A naive approach to avoiding attack windows is to "freeze" a package version, effectively never updating a codebase. While this approach is supported in some package ecosystems (such as specifying the `–save-exact` flag for the npm frontend's `install` command), it has the obvious detriment that security fixes and code improvements will never be incorporated via updates. The amount of risk a developer can tolerate from latent vulnerabilities versus malicious updates is likely to depend upon the particular stance of a given developer. In this work, our approach is to present the data on historical malicious updates, including an analysis of the severity of the attack.

4 Methodology

In this section, we present our methodology for analyzing malware security advisories in package ecosystems. Our statistical analysis focuses on the npm registry because of the prevalence of reports within that ecosystem. However, we verify that other registries follow roughly similar characteristics, even given the low number of total incidents reported.

4.1 GitHub Advisory Database

The GitHub Advisory Database provides a list of security advisories, collected from several sources, reviewed and republished by GitHub [22]. The security advisories are described by a JSON file in the Open Source Vulnerability format [6]. Advisories can be filtered by their relevant ecosystem, severity, Common Weakness Enumeration (CWE), and review status.

Due to the accessible nature of the GitHub Advisory Database and the practicality of its API access, we will use it as the main source of data for this study. We focus primarily on the npm ecosystem and the GitHub reviewed advisories. Of particular interest to us is CWE 506, defined as "contains code that appears to be malicious in nature" [40]. We use CWE 506 as the primary advisory of focus, as it most closely relates to the type of attack we study.

4.2 Popularity of Packages

Having over 3 million packages, npm is the largest package registry among similar ecosystems. The vast majority of those packages see little to no downloads, while a small percentage sees weekly

download counts in the hundreds of millions. We consider weekly download counts to be a proxy for the actual usage of a package. Thus, we define *popularity* by measuring the weekly download count.

Packages with high popularity have the potential to impact significantly more developers in the npm ecosystem compared to the majority of packages with low popularity. Due to this difference in potential impact, we consider it important to factor popularity into any analysis involving packages on npm. We collect download counts directly from npm by querying their API. Due to limitations of the API, we are unable to collect historical download counts of specific versions of packages. However, we are able to get historical download counts for all versions of a package. Despite this limitation, we believe that the download count of the package still acts as a good proxy for usage and our definition of popularity [39].

We conducted several measures of download counts. As npm displays the current weekly download counts of a package, we collected the current weekly download count for our set of packages. This is a crude measure and, while convenient to collect, does not align with the historical nature of the advisories we analyze. As an alternative, we could instead calculate the download count of a package for 1 week starting from when the malicious version was uploaded. However, due to the variability in the duration that the malicious versions exist, this weekly download count is not necessarily proportional to the time the malicious version existed. We also collected download counts for the entire period from the upload of the malicious version to the publication of the advisory. Although still an imperfect measure, we believe using the entire period from upload to advisory is a reasonable time frame based on the data available from the advisories and the download counts we are able to collect from npm. Despite the differences in time frame and period between these measures of download count, we did not observe significant deviations in our findings, nor did it change the conclusions we draw.

4.3 Research Questions

Our work seeks to build clarity around the use of delays in order to avoid downloading malicious versions of packages. Based on this goal, we formulate a primary research question as follows:

> How long should a package be present within a package registry before it is used?

We note that this research question depends on the risk profile of an individual developer or developer's organization. Consequently, we break down this question based on a number of factors:

- *RQ1*: What is the distribution of detection time for malicious packages across all advisories?
- *RQ2*: Does the severity of a malicious update attack correlate with its detection time?
- *RQ3*: Does the popularity of a package correlate with its detection time?

These questions provide additional structure with which a developer can set their own optimal minimum release delay. If a user has a risk profile that is highly sensitive to exposure to *any* malicious update, they may be interested primarily in the results of RQ1. If the user is only concerned with avoiding critical vulnerabilities, they might be most interested in RQ2. If the user does not expect to be using particularly obscure codebases that may skew the statistics, they may focus on RQ3. We explore these aspects of our study below.

5 Evaluation

From our analysis of security advisories, we present our findings on the detection time of malware on npm. In section 5.1 we examine the unweighted cumulative distribution of detection times. Next, in section 5.2 we break the advisories apart by severity. Then, in section 5.3 we weight the distribution by package popularity. Lastly, in section 5.4 we look at high profile advisories.

5.1 Distribution of Detection Time

At the time of our analysis, 14 June 2026, there were 6,768 advisories for npm in the GitHub Advisory Database. Of those, we analyze the 403 advisories with CWE-506. An advisory may include multiple identified packages and multiple versions of those packages. We identify 520 unique package and version combinations from the advisories.

To complete the analysis, the publish time of the identified version is required. Despite these versions having been removed from npm, for some packages, the publish time is still accessible from the npm registry API. Only 226 of the 520 package versions have that data available. We consider those package versions for our analysis.

For each package version, we calculate the time between its npm publication timestamp and the GitHub advisory's publication timestamp. As an advisory publication likely occurs after the initial detection or package removal, it is only a proxy for detection time. Figure 3 shows an empirical cumulative distribution plot of the detection time. We highlight the first 10 days in fig. 4 on a standard scale. There are several significant points along the cumulative distribution where we observe the detection of a significant percentage of malware, which have been marked in the figure.

Importantly, we find that 50% of malware is detected in approximately 18 hours and that 69% is detected in approximately 8 days. The next large milestones for detection percentage are 88.5% after approximately 465 days and 99.1% after approximately 893 days. We find this to be a significant jump in the persistence of malicious packages within the ecosystem.

5.2 Detection Time by Severity

Another lens through which we analyze advisories is severity. The advisories on the GitHub Advisory Database include a severity level of low, moderate, high, or critical based on the Common Vulnerability Scoring System (CVSS) score of the advisory. As advisories may be for multiple packages and versions, we assign each package and version the severity level of the advisory it is a part of. By separating the advisories by severity, we see that the vast majority are rated high or critical. The full breakdown by severity can be found in fig. 5.

Interestingly, there are 9 packages with low severity and 7 packages with medium severity. All packages with low severity are part of the same advisory "Broken dropper in @mistralai/mistralai, @mistralai/mistralai-azure, @mistralai/mistralai-gcp" [21]. The packages with medium severity are spread across these 5 advisories:

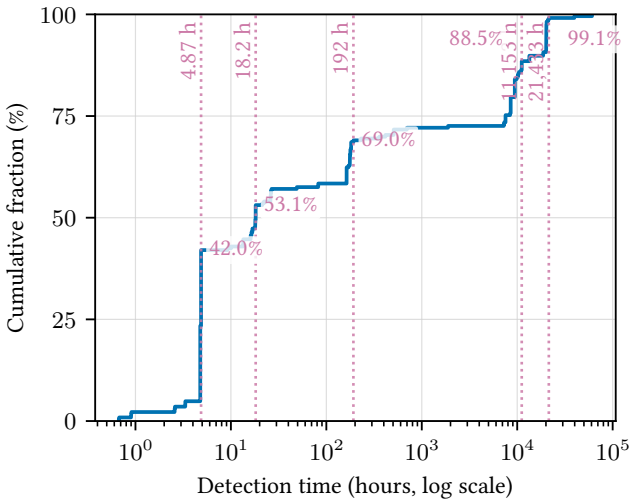


Figure 3: Cumulative Distribution of Detection Time

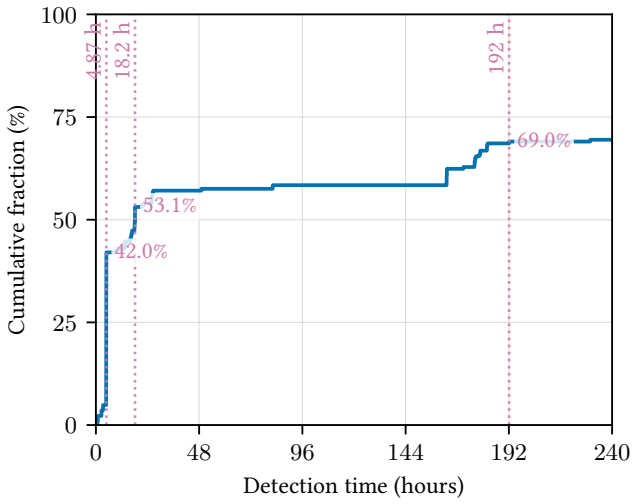


Figure 4: First 10 days of Detection Time Distribution

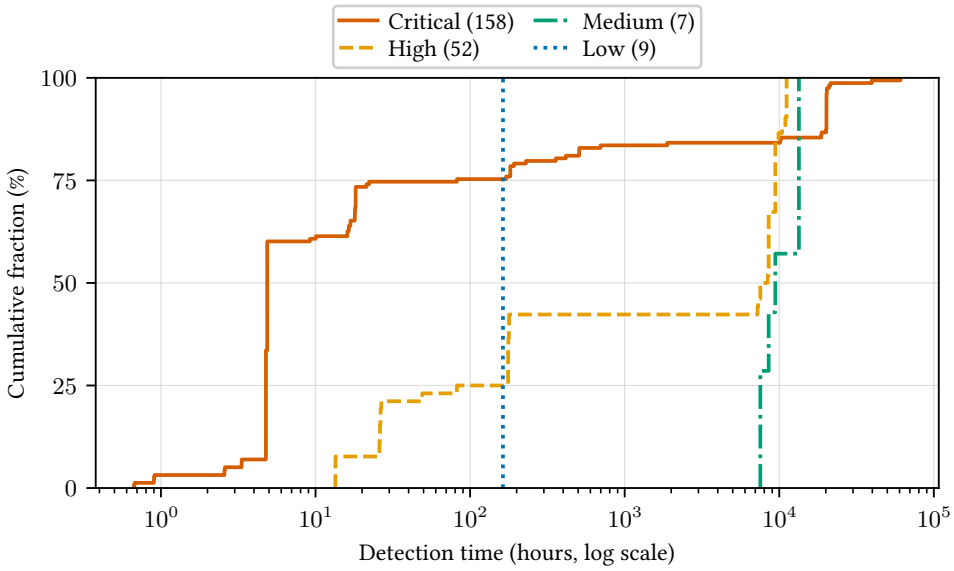


Figure 5: Cumulative Distribution of Detection Time by Severity

"MetaMask SDK indirectly exposed via malicious debug@4.4.2 dependency", "Hijacked Environment Variables in proxy.js", "jquey is malware", "cofeescript is malware", and "sqlite.js is malware" [12–15, 20].

In the case of the Mistral packages, the severity is given as low due to the presence of, but non-execution of, a malicious payload. The payload is in a different file than the one called for execution, and the calling code produces an error due to the incorrect file reference.

The first medium severity advisory for the MetaMask SDK packages is unique in that one of its dependencies was compromised. That dependency being version 4.4.2 of the debug package. There is a withdrawn advisory for the debug package [19]. The withdrawn advisory is not included in the set of advisories we analyzed. However, there is another advisory of high severity for version 4.4.2 of the debug package that was included. A severity level of medium was assigned due to specific conditions required to download the malicious debug version.

The remaining medium severity advisories were created before the existence of the GitHub Advisory Database. They were imported, reviewed, and published to the GitHub Advisory Database at a later date. We find it likely that packages containing malware were remediated before the advisory was published to the GitHub Advisory Database. These advisories contain no CVSS score, although GitHub labels them as medium severity. Instead, the advisories report an Exploit Prediction Scoring System (EPSS) score

of 1.123% for each advisory. We find it likely that the severity is classified as medium, as the malware had already been remediated, which is reflected in the EPSS score.

5.3 Detection Time by Popularity

The results of figs. 3 and 5 imply mixed results for a delay window policy. Developers may need only delay by 0.76 days to exceed the average vulnerability window for all packages on npm (and similarly to avoid critical vulnerability windows). However, avoiding 99% of vulnerability windows requires an impractically-long delay window.

While these findings are important, we note that they disregard an important reality of package ecosystems and npm specifically. The actual *use* of packages in npm is highly skewed, see fig. 6. In practice, a small fraction of packages is responsible for nearly all the weekly downloads from the registry. Conversely, well over 80% of the packages in npm have such low download counts that they are effectively unused. Indeed, npm allows for the unpublishing of packages with fewer than 300 weekly downloads, and previous work has identified a threshold of 350 weekly downloads to be effectively unused [29, 39]. Thus, any package displaying a download count below 350 is extremely unlikely to be integrated into a third-party codebase.

Treating all packages as equally likely to be downloaded has the potential to mislead developers about the likelihood that their own package dependency installations and updates will fall into a vulnerability window. If popular packages tend to take longer to detect than unpopular packages, an unweighted analysis will underestimate the delay relevant to typical downloads. Conversely, if unpopular packages persist longer because they receive less scrutiny, an unweighted analysis will overestimate the delay relevant to typical downloads.

We find popularity to be an important measure in any analysis of data related to packages on npm. More popular packages have the potential to impact significantly more people compared to less popular packages. Due to the distribution of package popularity, this potential impact ranges across several orders of magnitude.

In fig. 7 we weight the distribution of detection windows by the packages' weekly download counts. We note a single dramatic increase ending at approximately 8 days with a weighted 94.2% detection. Figure 8 shows the first 10 days in linear scale for this critical section. Compared to the unweighted cumulative distribution, there is a difference of approximately 30 percentage points within the same time frame of approximately 8 days.

We find that the most long-lived vulnerabilities appear inversely correlated with package popularity. This finding supports previous work showing that code scrutiny is an important factor in discovering vulnerabilities [41].

5.4 High Profile Advisories by Popularity

We found advisories that cover 82 packages with over 1 million downloads during the period between their upload to npm and the publication of the advisory. These packages are of particular interest due to their high potential impact. We include analysis for three of those packages with over 100 million downloads.

All three packages were part of the same malware campaign that started with the compromise of the packages' author through a phishing attack. On 8 September 2025, npm user *qix* was compromised through a fake 2FA reset email and had several of their packages compromised. We focus on the three packages with over 100 million downloads: `debug`, `color-convert`, and `color-name`. There are three GitHub Advisories for the three packages that were published on 15 September 2025 [16–18]. The malware was detected on 8 September 2025 and reported independently by Aikaido Security, Ox Security, and Socket [5, 9, 23].

New versions of those three packages were released that were identical to the previous versions but included malware. The included malware was intended to operate in a browser environment and redirected cryptocurrency transactions to the attacker. The compromised versions of the packages were removed from npm on 8 September 2025. On 13 September 2025, the maintainer uploaded a new version of each package that was identical to the previous clean version without the malware. We find this to be an ideal example of our conceptual timeline from fig. 2.

6 Discussion

In this section, we discuss the findings of our work, give our commentary on the minimum release age feature, and address perceived issues.

6.1 Summary of Results

In our analyzed sample, 94.2% of the download-weighted detection time falls at approximately 8 days, as shown in fig. 7. We therefore identify 8 days as a potential recommended minimum release age threshold.

We note that this finding demonstrates the importance of breaking down statistical evaluation based on the factors that impact real decision making in package ecosystems. In particular, taking aggregate statistics over all packages is likely to direct developers to an unsatisfying conclusion: in order to avoid 90% of the incident windows on npm, a user would need to wait over 1000 days, as per fig. 3. Obviously, this delay would be impractical. However, when factoring in the additional context of the popularity of packages, the situation becomes much more manageable with 94.2% detected after 8 days when weighted by popularity.

We provide this recommendation of 8 days as an actionable result that should work well for most developers. We also acknowledge that some developers or organizations might have a risk profile that induces a different delay window. For example, fig. 5 shows that the vast majority of vulnerabilities under CW 506 are critical vulnerabilities, and those critical vulnerabilities shape the overall distribution of vulnerability windows. Consequently, a developer cannot get away with a shorter delay window simply by disregarding exposure to low, medium, or even high-severity vulnerabilities.

6.2 Commentary on Minimum Release Age

The ability to set a minimum release age has seen wide adoption by frontends in the npm ecosystem, including npm, pnpm, and Yarn. This wide adoption presents some indication that the maintainers of these package managers see the feature as useful or at least recognize that some users of their software desire the feature. Our

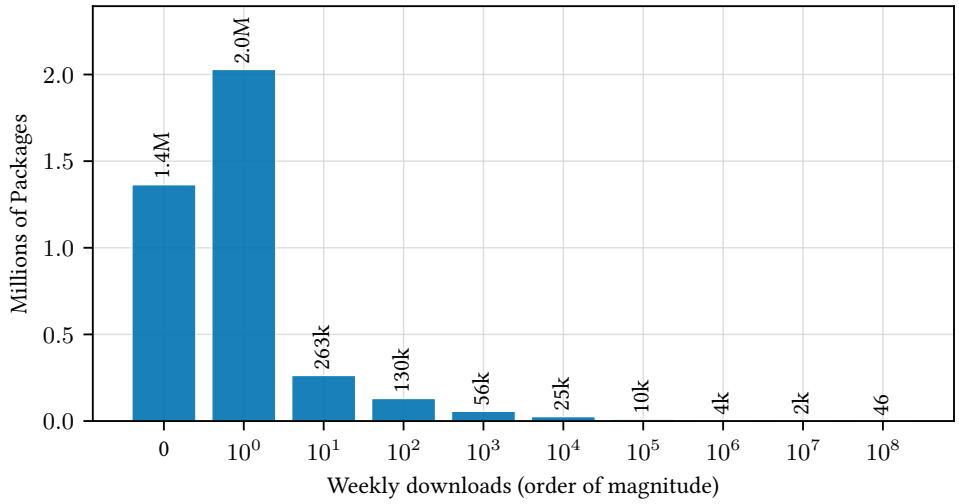


Figure 6: Weekly Download Counts of all npm Packages

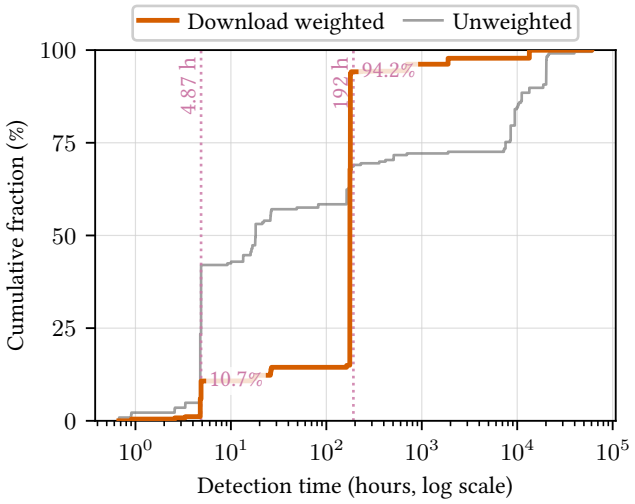


Figure 7: Download-Weighted Detection Time Distribution

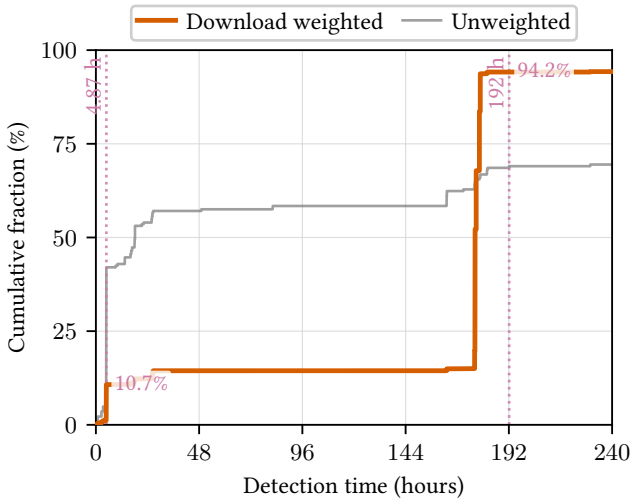


Figure 8: Download-Weighted Detection Time Distribution, First 10 Days

work, as presented, is an attempt to find evidence-based justification for this feature.

We find the cost of setting a minimum release age to be minor. The ability to set and forget the feature means that it does not contribute to a developers cognitive load when installing packages. As this feature works invisibly during package installation, a developer would only be impacted in the situation where they explicitly intend to install the absolute latest version of a package within the set minimum release age. We believe this to be a relatively infrequent occurrence and would likely be minimally impactful if it did coincidentally occur.

One scenario we conceive in which the minimum release age feature could present an issue is if a user urgently needs a package update that was just released. This could occur if a package on which a user depends has a critical bug or vulnerability and must be updated to ensure the integrity of their software. While we recognize that such a situation may arise, we still believe that users should exercise due diligence when updating so as not to cause potentially more severe issues. In a case where a user has taken proper precautions and the best option is to update, the minimum release age could be temporarily relaxed or removed, or the specific package could be excluded from the minimum release age. As this update is intentional on the user's part, we believe this is not too burdensome.

6.3 Who Finds the Malware

A relevant observation is that even if developers wait until a package is x amount of time old to install it, that does not mean the malware will not be found until x amount of time.

Indeed, while developers would be the ones waiting x amount of time, security labs and researchers will not. As we saw in the phishing attack detailed in section 5.4, which involved malware introduced to the *debug* package, among others, it was three security labs that independently detected and reported the malware. There is an incentive for others to find malware without needing to have developers on the front lines reporting it after having been compromised.

We find it important to note that this is not a panacea. Even if a user were to set an extraordinarily high minimum release age such that they would have avoided all the malware from the advisories we analyzed, it is still theoretically possible that they could download a package with malware. Setting the minimum release age to a reasonable time frame based on the findings in our analysis is just one of many measures a developer can take to secure themselves against malware. This feature is not a replacement for performing due diligence before installing a package from npm.

7 Future Work

The work presented herein provides a concrete answer to our research question that we hope will be adopted as best practice for setting a package release age. Nevertheless, there are additional factors that require further attention. We suggest studying these factors in future work.

Vulnerability Analysis: Our work focuses on the benefits of update delay. However, as described in section 3, there are detriments to waiting for a package version to meet a release age before it is integrated into a codebase. Setting a minimum release age can also delay urgent security fixes. Future work should, therefore, study whether security releases can be identified reliably and exempted from the delay without creating an exemption mechanism that attackers can abuse. The system would need to be robust against an adversary uploading a malicious version with a critical bug fix tag. The complexity of this issue motivates us to consider such a study as an area for future research.

Ecosystem Scope: We believe that our approach transfers across package ecosystems, providing a rough guideline to avoid attack windows in general. However, we note that security analysis and community standards may have implications for the optimal minimum release age in a given ecosystem. In this work, we limit our analysis to package ecosystem reports in the GitHub Security Advisory database, which contains relatively few incidents for ecosystems besides npm. Nevertheless, it may be possible to collect incident reports from other sources to enrich our analysis here.

Feedback Loops: A potential concern that may arise if developers commonly adopt a minimum release age is that it will lessen the scrutiny of a package in the immediate wake of its release. One might imagine a scenario where, if a package is not used until 8 days after it is released, then there are fewer opportunities to uncover malicious activities, thus driving up the minimum release age. Taken to a logical extreme, this situation would induce a feedback loop where developers would wait ever longer to download and update packages in the hope that others would encounter malicious updates first.

While we are interested in studying how a commonly-accepted minimum release age impacts the optimal minimum release age, it is unlikely that such a feedback loop will occur in practice. Many of the malicious version reports that we discover in the GitHub Advisory Database were uploaded not by developers falling victim to a malicious package version, but rather by security analysts specifically searching for such code and reporting it prior to use. As such, lengthening the common minimum release age is unlikely to reduce the scrutiny on a package version while giving security analysts more time to uncover incidents. Nevertheless, future studies of the interplay between minimum release age and time to discovery are required.

Reactive Adversaries: Similar to the above point, a common delay might incentivize attackers to behave differently. For example, an adversary might choose to inject a "timebomb" attack that does not deploy its payload until after the common delay has passed, with the hope of increasing the impact of the attack and evading detection by analysts. It is unclear whether there is a benefit to the attacker in hiding a payload this way. Similarly, it is unclear whether the mechanisms used to detect malicious code in package ecosystems are sensitive to a delay window (for instance, static analysis is unlikely to be impacted by a delay window injected directly into the code). Fully capturing how detection is currently performed and how adversaries react to update delays is an interesting future direction.

Historical Analysis of Removed Packages: As described in section 4.1, our analysis is based primarily on publicly-available npm registry metadata. In particular, we rely on package publication dates to determine when a malicious update is introduced into a package's timeline. This analysis is complicated by npm's policy of removing known-malicious code from the registry, which often renders the associated metadata inaccessible through the public interface. We would be interested in reproducing our analysis with the full results of all package takedowns, including those that are not accessible. Nevertheless, we cannot discern a pattern among those packages whose metadata has been removed, so we do not have a strong reason to believe that these packages would skew our results.

8 Related Work

The importance of software supply chain security has been recognized for years, and numerous studies have examined various aspects of the problem. Here, we discuss the works most relevant to our study, presenting representative examples and contrasting them with our own work.

Software Supply Chain Attacks and Package Ecosystem Risks: The nature of code sharing using package ecosystems makes software supply chain attacks a reality that developers must manage. Ladisa et al. provide a taxonomy covering attacks against code repositories, build systems, distribution infrastructure, maintainers, and package users [26]. To complement this taxonomy, Ohm et al. conduct an empirical analysis of a collection of malicious packages collected from several package ecosystems, which they call the

Backstabber's Knife Collection [31]. For npm specifically, Zimmermann et al. show how a compromise of a relatively small number of packages can affect a disproportionately large section of the ecosystem due to the highly connected dependency graph. Zahan et al. provide an analysis of package and maintainer characteristics that contribute to weak links in the npm supply chain [42]. These studies motivate defenses against malicious package releases and demonstrate that the consequences of a compromise depend not only on the malicious package itself but also on its position in the dependency and maintainer networks.

Malicious Package Detection and Containment: There is a substantial body of work on techniques for the detection of malicious packages. Garrett el al. investigate changes between package versions to identify suspicious and potentially malicious updates to packages on npm [11]. Vu et al. and Scalco et al. examine the differences between source and package to identify malicious PyPI and npm packages, respectively [35]. DONAPI constructs behavioral representations for detecting malicious npm packages, and Zhang et al. investigate a shared malicious-behavior model for npm and PyPI [24, 43]. These systems contribute to the scrutiny that may cause a malicious version to be identified during the interval measured in our study.

Other work investigates a different and complementary approach to detection, limiting the consequences of executing malicious code. Ferreira et al. explore a runtime permission system that enforces restrictions on sensitive resources [10].

Discovery and Disclosure Timelines: The effectiveness of the minimum release age feature is dependent on how quickly malicious package releases are discovered and remediated. Alfadel et al. examine the discoverability of npm vulnerabilities and show that dependency vulnerabilities can remain undisclosed for extended periods [2]. Wyss et al. present evidence that scrutiny of a codebase is the best predictor that bugs will be found in a timely manner [41].

Cost of Delaying Updates: Delaying package update adoption can reduce exposure to newly introduced malware, but it can also postpone the adoption of benign improvements and vulnerability fixes. Prior work shows that dependency updates are already subject to substantial delays. Kula et al. find that developers frequently continue to use outdated library versions and do not always migrate promptly in response to security advisories [25]. Decan et al. characterize this phenomenon as technical lag and analyze how dependency constraints and release practices affect the distance between installed and available npm versions [8]. Chinthanet et al. focus specifically on npm vulnerability fixes and identify separate delays in the release, downstream adoption, and transitive propagation of fixed versions [7]. They also show that security fixes are commonly bundled with unrelated changes, complicating attempts to exempt security-only releases from an age policy.

Measuring Ecosystem Exposure and Impact: Because package use is highly skewed, several studies caution against treating all packages and dependencies as equally consequential. Abdalkareem et al. analyze the use and impact of trivial packages in npm and PyPI [1]. Zimmermann et al. demonstrate that dependency-network position creates substantial variation in the number of downstream projects potentially affected by a package [44]. Taylor et al. use download-related popularity signals when assessing typosquatting risk [39]. These works motivate our use of package popularity to supplement an unweighted analysis of malicious-release windows.

Downloads provide only an approximate measure of exposure. Latendresse et al. distinguish declared or installed npm dependencies from dependencies actually used in production and find that many installed dependencies are not used in deployed code [27]. Pashchenko et al. similarly show, in the context of vulnerable dependencies, that package presence can overestimate real exposure when the affected functionality is not reachable [33]. Our download-weighted results should therefore be interpreted as an estimate of potential installation exposure rather than a direct measure of executed malicious code or realized compromise. Download counts remain useful because they capture large differences in package adoption that an unweighted analysis would ignore.

Prior work has characterized malicious package detection and its relationship to ecosystem impact and popularity. It has further proposed mechanisms for detecting or containing malicious packages. Our work builds on these measurements and focuses on the newly introduced minimum release age feature. We use historical malware advisories to inform the configuration of this feature.

9 Conclusion

The prevalence of software supply chain attacks against the package ecosystem motivates the need to mitigate the impacts of malicious code in packages. Mechanisms, both technical and social, have been implemented to mitigate these impacts. The effectiveness of these mechanisms needs to be validated through evidence-based analysis. We believe the minimum release age feature implemented in npm, pnpm, Yarn, and other frontends is worthy of study.

In this paper, we study the use of the minimum release age feature to mitigate the impact of malicious updates. The minimum release age feature has been introduced into package frontends as a mitigation strategy that aims to avoid the impact of a malicious update by waiting until after the package has been appropriately scrutinized. This feature relies on setting a specific time for the minimum release age. We conduct an analysis of the GitHub Advisory Database to develop guidance around finding the optimal minimum release age.

To our knowledge, our work is the first study focused on estimating candidate ages for the minimum release age feature. We consider the effectiveness of a delay in exceeding the length of an attack by studying several dimensions, such as the average response time for the attack, the correlation of an attack window to the severity of the incident, and the likelihood that a given package related to an incident is downloaded and integrated into a developer's codebase. We believe these analyses provide developers with the necessary information needed to choose a minimum release age that meets their security requirements.

We find that the inclusion of popularity in our analysis is particularly effective in providing actionable value for a delay: we find that a delay of 8.01 days yields a 94.2% chance that a user can avoid any malicious update to which they would be exposed.

References

[1] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the Impact of Using Trivial Packages: An Empirical Case Study on npm and PyPI.

Empirical Software Engineering 25, 2 (March 2020), 1168–1204. doi:10.1007/s10664-019-09792-9

[2] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. 2023. On the Discoverability of npm Vulnerabilities in Node.js Projects. *ACM Transactions on Software Engineering and Methodology* 32, 4, Article 91 (May 2023), 27 pages. doi:10.1145/3571848

[3] Merav Bar and Rami McCarthy. 2025. s1ngularity: supply chain attack leaks secrets on GitHub: everything you need to know | Wiz Blog. https://www.wiz.io/blog/s1ngularity-supply-chain-attack

[4] Merav Bar, Rami McCarthy, and Barak Sharoni. 2025. Shai-Hulud npm Supply Chain Attack | Wiz Blog. https://www.wiz.io/blog/shai-hulud-npm-supply-chain-attack

[5] Moshe Siman Tov Bustan and Alexander Chailytko. 2025. 19 npm Packages Compromised in Major Supply-Chain Attack. OX Security. Published September 8, 2025; accessed July 9, 2026. https://www.ox.security/blog/npm-packages-compromised/

[6] Oliver Chang and Russ Cox. 2026. Open Source Vulnerability format. publisher: Open Source Security Foundation. https://ossf.github.io/osv-schema/

[7] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the Release, Adoption, and Propagation of npm Vulnerability Fixes. *Empirical Software Engineering* 26, 3, Article 47 (March 2021), 28 pages. doi:10.1007/s10664-021-09951-x

[8] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Piscataway, NJ, USA, 404–414. doi:10.1109/ICSME.2018.00050

[9] Charlie Eriksen. 2025. npm debug and chalk Packages Compromised. Aikido Security. Published September 8, 2025; accessed July 9, 2026. https://www.aikido.dev/blog/npm-debug-and-chalk-packages-compromised

[10] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2021. Containing Malicious Package Updates in npm with a Lightweight Permission System. In *43rd IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE, Piscataway, NJ, USA, 1334–1346. doi:10.1109/ICSE43902.2021.00121

[11] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting Suspicious Package Updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, Piscataway, NJ, USA, 13–16. doi:10.1109/ICSE-NIER.2019.00012

[12] GitHub. 2018. CVE-2017-16050: sqlite.js Is Malware. GitHub Advisory Database. Published July 23, 2018; accessed July 8, 2026. https://github.com/advisories/GHSA-6f34-pq9f-36vf

[13] GitHub. 2018. CVE-2017-16076: Hijacked Environment Variables in proxy.js. GitHub Advisory Database. Published August 29, 2018; accessed July 8, 2026. https://github.com/advisories/GHSA-qj73-v688-wqxf

[14] GitHub. 2018. CVE-2017-16202: cofeescript Is Malware. GitHub Advisory Database. Published August 6, 2018; accessed July 8, 2026. https://github.com/advisories/GHSA-m6wh-m8m8-6xx5

[15] GitHub. 2018. CVE-2017-16204: jquey Is Malware. GitHub Advisory Database. Published August 6, 2018; accessed July 8, 2026. https://github.com/advisories/GHSA-6fjr-m7v6-fpg9

[16] GitHub. 2025. CVE-2025-59144: debug@4.4.2 Contains Malware After npm Account Takeover. GitHub Advisory Database. Published September 15, 2025; accessed July 9, 2026. https://github.com/advisories/GHSA-4x49-vf9v-38px

[17] GitHub. 2025. CVE-2025-59145: color-name@2.0.1 Contains Malware After npm Account Takeover. GitHub Advisory Database. Published September 15, 2025; accessed July 9, 2026. https://github.com/advisories/GHSA-5fvm-p68v-5wmh

[18] GitHub. 2025. CVE-2025-59162: color-convert@3.1.1 Contains Malware After npm Account Takeover. GitHub Advisory Database. Published September 15, 2025; accessed July 9, 2026. https://github.com/advisories/GHSA-pxx3-g568-hxr4

[19] GitHub. 2025. GHSA-8mgj-vmr8-frr6: Duplicate Advisory: Malware in debug. GitHub Advisory Database. Published September 8 and withdrawn September 15, 2025; accessed July 8, 2026. https://github.com/advisories/GHSA-8mgj-vmr8-frr6

[20] GitHub. 2025. GHSA-qj3p-xc97-xw74: MetaMask SDK Indirectly Exposed via Malicious debug@4.4.2 Dependency. GitHub Advisory Database. Published September 15, 2025; accessed July 8, 2026. https://github.com/advisories/GHSA-qj3p-xc97-xw74

[21] GitHub. 2026. GHSA-jgg6-4rpr-wfh7: Broken Dropper in @mistralai/mistralai, @mistralai/mistralai-azure, and @mistralai/mistralai-gcp. GitHub Advisory Database. Published May 18, 2026; accessed July 8, 2026. https://github.com/advisories/GHSA-jgg6-4rpr-wfh7

[22] GitHub. 2026. GitHub Advisory Database. GitHub Docs. Accessed July 1, 2026. https://docs.github.com/en/code-security/concepts/vulnerability-reporting-and-management/github-advisory-database

[23] Sarah Gooding, Olivia Brown, Kush Pandya, Philipp Burckhardt, Peter van der Zee, and Douglas Coburn. 2025. npm Author Qix Compromised via Phishing Email in Major Supply Chain Attack. Socket. Published September 8, 2025; accessed July 9, 2026. https://socket.dev/blog/npm-author-qix-compromised-in-major-supply-chain-attack

[24] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM Packages Detector Using Behavior Sequence Knowledge Mapping. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3765–3782. https://www.usenix.org/conference/usenixsecurity24/presentation/huang-cheng

[25] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. doi:10.1007/s10664-017-9521-5

[26] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, Piscataway, NJ, USA, 1509–1526. doi:10.1109/SP46215.2023.10179304

[27] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not All Dependencies Are Equal: An Empirical Study on Production Dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. doi:10.1145/3551349.3556896

[28] Ilyas Makari. 2026. Red Hat npm Packages Compromised to Spread a Credential-Stealing Worm. https://www.aikido.dev/blog/red-hat-npm-packages-compromised-credential-stealing-worm

[29] npm. 2026. npm Unpublish Policy | npm Docs. https://docs.npmjs.com/policies/unpublish/

[30] npm. 2026. Release v11.10.0. Released February 11, 2026; accessed July 2, 2026. https://github.com/npm/cli/releases/tag/v11.10.0

[31] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. 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 (Lecture Notes in Computer Science, Vol. 12223)*. Springer, Cham, Switzerland, 23–43. doi:10.1007/978-3-030-52683-2_2

[32] Kush Pandya, Peter van der Zee, Olivia Brown, and Socket Research Team. 2025. Updated and Ongoing Supply Chain Attack Targets CrowdStrike ... https://socket.dev/blog/ongoing-supply-chain-attack-targets-crowdstrike-npm-packages

[33] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2022. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* 48, 5 (May 2022), 1592–1609. doi:10.1109/TSE.2020.3025443

[34] pnpm contributors. 2026. Mitigating Supply Chain Attacks. pnpm Documentation. Accessed July 2, 2026. https://pnpm.io/supply-chain-security

[35] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. 2022. On the Feasibility of Detecting Injections in Malicious npm Packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*. Association for Computing Machinery, New York, NY, USA, Article 115, 8 pages. doi:10.1145/3538969.3543815

[36] Socket Research Team. 2026. Mini Shai-Hulud Campaign Hits Red Hat Cloud Services npm Pac... https://socket.dev/blog/mini-shai-hulud-campaign-hits-red-hat-cloud-services-npm-packages

[37] Socket Research Team. 2026. Mini Shai-Hulud Hits @antv Ecosystem, 639 Compromised npm Pa... https://socket.dev/blog/antv-packages-compromised

[38] Socket Research Team. 2026. TanStack npm Packages Compromised in Ongoing Mini Shai-Hulud... https://socket.dev/blog/tanstack-npm-packages-compromised-mini-shai-hulud-supply-chain-attack

[39] Matthew Taylor, Ruturaj K. Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *Network and System Security: 14th International Conference, NSS 2020 (Lecture Notes in Computer Science, Vol. 12570)*. Springer, Cham, Switzerland, 112–131. doi:10.1007/978-3-030-65745-1_7

[40] The MITRE Corporation. 2026. CWE-506: Embedded Malicious Code, Version 4.20. Accessed July 1, 2026. https://cwe.mitre.org/data/definitions/506.html

[41] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2023. (Nothing But) Many Eyes Make All Bugs Shallow. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. Association for Computing Machinery, New York, NY, USA, 53–63. doi:10.1145/3605770.3625216

[42] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What Are Weak Links in the npm Supply Chain?. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. Association for Computing Machinery, New York, NY, USA, 331–340. doi:10.1145/3510457.3513044

[43] Junan Zhang, Kaifeng Huang, Yiheng Huang, Bihuan Chen, Ruisi Wang, Chong Wang, and Xin Peng. 2025. Killing Two Birds with One Stone: Malicious Package Detection in NPM and PyPI Using a Single Model of Malicious Behavior Sequence. *ACM Transactions on Software Engineering and Methodology* 34, 4 (April 2025), 1–28. doi:10.1145/3705304

[44] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman