TLP - CLEAR

EUROPEAN UNION AGENCY
FOR CYBERSECURITY

ENISA Technical Advisory for Secure Use of Package Managers

MARCH 2026

About ENISA

The European Union Agency for Cybersecurity, ENISA, is the Union's agency dedicated to achieving a high common level of cybersecurity across Europe. Established in 2004 and strengthened by the EU Cybersecurity Act, the European Union Agency for Cybersecurity contributes to EU cyber policy, enhances the trustworthiness of ICT products, services and processes with cybersecurity certification schemes, cooperates with Member States and EU bodies, and helps Europe prepare for the cyber challenges of tomorrow. Through knowledge sharing, capacity building and awareness raising, the Agency works together with its key stakeholders to strengthen trust in the connected economy, to boost resilience of the Union's infrastructure, and, ultimately, to keep Europe's society and citizens digitally secure. More information about ENISA and its work can be found here: www.enisa.europa.eu.

CONTACT

To contact the authors, use product_security@enisa.europa.eu.
For media enquiries about this paper, use press@enisa.europa.eu.

AUTHORS

ENISA

ISBN 978-92-9204-788-7, ISSN 3094-600X
doi:10.2824/6157993, Catalogue number TP-01-26-005-EN-N

Table of Contents

1. Scope and context

Modern software development relies heavily on external packages, libraries and tools obtained through package managers such as the *npm* registry ([1]), package installer for Python (*pip*) ([2]) and *Apache Maven* ([3]) repositories. These package managers support software development by providing quick and easy access to vast repositories of third-party code, and often open-source code. However, as recent events have demonstrated ([4]) ([5]) ([6]), they also introduce supply chain risks ([7]). This document focuses on how developers can securely use package managers as part of their software development life cycle. In particular, this document:

- outlines common risks involved in the use of third-party packages;
- presents secure practices for selecting, integrating, and monitoring packages; and
- describes approaches for addressing vulnerabilities found in dependencies.

The scope of this document is limited to the security considerations involved in consuming packages and managing dependencies within software projects, primarily at the application level. Operating system package managers, secure package publication, security of the package manager distribution channel and secure coding practices, while important, are outside the scope of this document.

Examples listed in this document often reference popular ecosystems and tools like NodeJS's *npm* package manager and code repository platforms such as *GitHub*, but the main principles apply across package ecosystems and code repositories.

1.1 Note on the current version of the document

The current version (v1.1) of the document has been released after the ENISA call for feedback ([8]) that ran between 17 December 2025 and 20 January 2026. As part of this process, ENISA received 15 additional contributions from public and private stakeholders, cybersecurity experts, software development practitioners and the open-source community. We would like to publicly thank all stakeholders that provided their views and feedback! All contributions have been reviewed and, to the maximum extent possible, the recommendations and observations have been addressed in this final version. This document may go through regular revisions.

([1]) NPM, https://www.npmjs.com/, accessed 28 November 2025.
([2]) PyPI, https://pypi.org/project/pip/, accessed 28 November 2025.
([3]) Maven, https://maven.apache.org/repositories/index.html, accessed 28 November 2025.
([4]) Palo Alto Networks, 'Breakdown: Widespread npm supply chain attack puts billions of weekly downloads at risk', 10 September 2025, https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/, accessed 28 November 2025.
([5]) Aikido, 'XRP supply chain attack: Official NPM package infected with crypto stealing backdoor', 22 April 2025, https://www.aikido.dev/blog/xrp-supplychain-attack-official-npm-package-infected-with-crypto-stealing-backdoor, accessed 28 November 2025.
([6]) Wiz, 'Shai-Hulud 2.0 supply chain attack: 25K+ repos exposing secrets', 24 November 2025, https://www.wiz.io/blog/shai-hulud-2-0-ongoing-supply-chain-attack, accessed 28 November 2025.
([7]) European Union Agency for Cybersecurity, *ENISA threat landscape for supply chain attacks*, 2021, https://data.europa.eu/doi/10.2824/168593, accessed 28 November 2025.
([8]) ENISA, 'Call for feedback: Advancing software supply chain security together!', press release, 17 December 2025, https://www.enisa.europa.eu/news/call-for-feedback-advancing-software-supply-chain-security-together, accessed 11 February 2026.

2. Solution architecture and data flows

2.1 How package managers work

Package managers automate the process of installing, updating, configuring, and removing software libraries and their dependencies. They are an important part of modern software development, enabling code reuse, consistent builds and simplified updates.

While implementations vary across ecosystems, the underlying principles remain the same, i.e. package managers interact with package repositories to retrieve packages, resolve dependencies and integrate them into applications. The table below provides examples of common package managers, their associated ecosystems and links to their official documentation.

Table 1: List of common package managers

Package manager	Ecosystem	Official link
npm	Node.js/JavaScript	https://docs.npmjs.com
yarn	Node.js/JavaScript	https://yarnpkg.com
Pip/PyPI	Python	https://pip.pypa.io/en/stable/
Conda	Python/R	https://docs.conda.io
Maven	Java	https://maven.apache.org/
Gradle	Java	https://docs.gradle.org
SPM	Swift	https://docs.swift.org/package-manager/
CocoaPods	Swift/Obj-C	https://guides.cocoapods.org
NuGet	.NET	https://learn.microsoft.com/nuget
RubyGems	Ruby	https://guides.rubygems.org
Composer	PHP	https://getcomposer.org
CPAN	Perl	https://www.cpan.org
CRAN	R	https://cran.r-project.org
Cargo	Rust	https://doc.rust-lang.org/cargo

2.1.1 Core elements

At a high level, package managers work through a set of key elements that allow developers to share, distribute and consume software packages.

- **Package.** A bundle of code that provides a specific functionality. The code is then packaged and uploaded to a package manager for reuse.
- **Dependencies.** Represent relationships between packages. Each package may rely on other packages, forming a dependency tree. Dependencies of dependencies are referred to as transitive dependencies.
- **Package developer.** The author or maintainer who creates and publishes packages or libraries for reuse by other applications or packages.
- **Application.** The end product or project that makes use of one or more packages to provide extended functionality.
- **Package repository.** A central registry (e.g. *npmjs.com*, *pypi.org*, *rubygems.org*) that hosts published packages and their metadata, allowing developers to search for, download and

manage them. The package repository acts as a distributor of the package, making it available and discoverable.

- **Application developer.** The consumer who integrates packages into their applications, relying on the package manager to resolve dependencies, manage versions and ensure consistency.
- **Package manager.** The underlying tool that facilitates the installation, configuration, update and removal of packages and their dependencies. It acts as the intermediary between developers and the package repository.

The diagram below illustrates how package developers publish reusable code modules to a shared repository, allowing application developers to download and integrate these packages into their own projects.

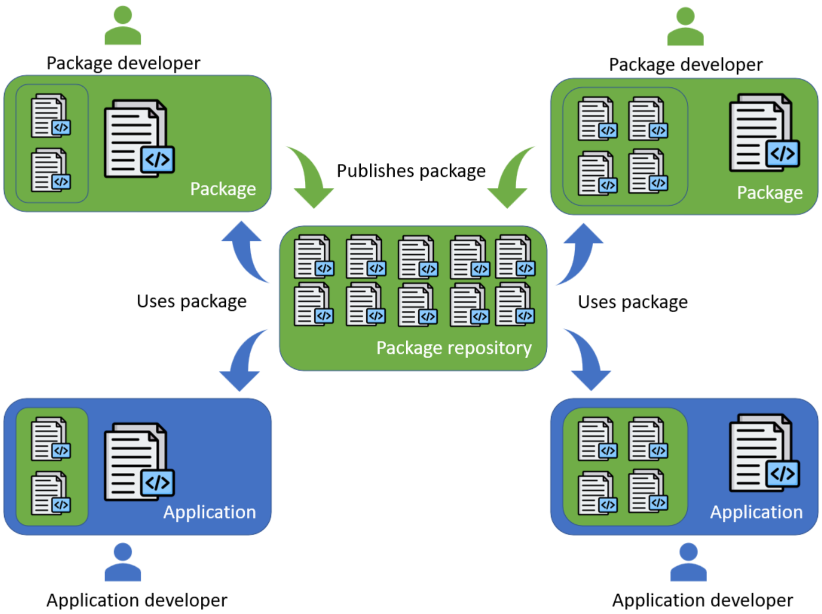

Figure 1: Package publishing and consumption

2.1.2 Example flow

The table below shows what happens when an application developer writes a small piece of code that makes use of a package (express) downloaded through the popular Node.js package manager npm.

<table>
<tr><th>Action</th><th>Example</th></tr>
<tr><td>Developer writes a simple Node.js app.</td><td><pre>
const express = require('express');
const app = express();

app.get('/', (req, res) => res.send('Hello from Express!'));

const port = 3000;
app.listen(port, () => {
  console.log(`Listening on http://localhost:${port}`);
});
</pre></td></tr>
</table>

Developer adds the express package to the Node.js project using *npm install express*.
 	
(npm downloads the package from the registry).
	`Node.js v18.14.0`
	`PS C:\Users\Documents\vscode_projects\simple_nodejs> npm install express`
	
	`added 68 packages, and audited 69 packages in 4s`

The package express itself depends on other packages (e.g. router, qs, depd).
 	
NB: the command *npm view <package> dependencies* shows the direct dependencies.
	`PS C:\Users\Documents\vscode_projects\simple_nodejs> npm view express dependencies`
	`>>`
	`{`
	`qs: '^6.14.0',`
	`etag: '^1.8.1',`
	`once: '^1.4.0',`
	`send: '^1.1.0',`
	`vary: '^1.1.2',`
	`debug: '^4.4.0',`
	`fresh: '^2.0.0',`
	`cookie: '^0.7.1',`

A lockfile (package-lock.json) records the exact versions of all installed dependencies, including transitive dependencies (packages that your project relies on indirectly).	{} node_modules/router version : "2.2.0" resolved : "https://registry.npmjs.org/router/-/router-2.2.0.tgz" integrity : "sha512-nLTrUKm2UyiL7rlhapu/Zl45FwNgkZGaCpZblHajDYgwlJCOzLSk+clPAnsEqV955GjlLJn {} dependencies debug : "^4.4.0" depd : "^2.0.0" is-promise : "^4.0.0" parseurl : "^1.3.3" path-to-regexp : "^8.0.0" {} engines

This process seems straightforward and simple, where with just 10 lines of code a developer can set up a web server by (re)using the popular *express* web-framework. However, it is important to understand what is happening in the background when the express package and dependencies are included in the project.

2.1.3 Package import and usage

Importing a package does not just add that single module. It also includes all of its direct and transitive dependencies. In our example, importing *express* brings in around **27** direct dependencies, but a total of **68** packages are installed once all transitive dependencies are resolved.

Ultimately, application developers need to accept that by installing a package and all of its dependencies, the project's dependency graph increases, together with the potential attack surface. Although commonly referred to as 'third-party', once included in your source these components become part of the trusted codebase. As such, they should receive security scrutiny similar to first-party code.

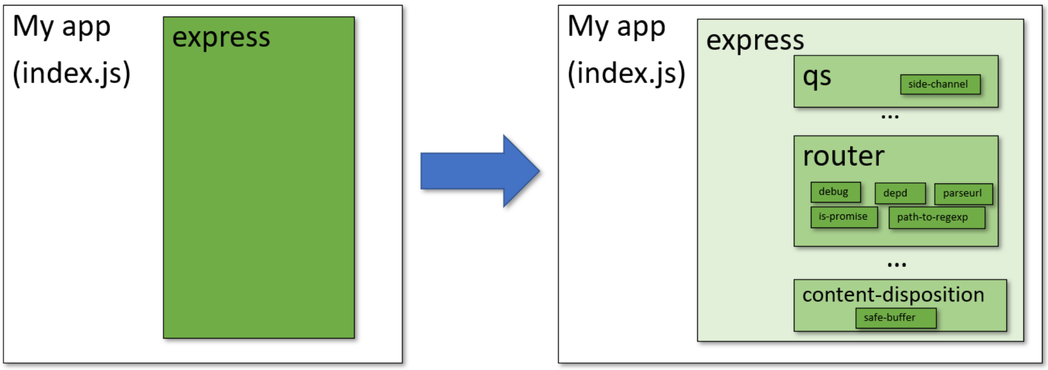

Figure 2: Package dependencies

However, it is important to note that importing a package does not mean all of its code is executed or even reachable at runtime. For example, when using *express* to only define a few basic routing endpoints ([9]), the application might only execute a small fraction of the package's total code.

Yet, even when only these basic routing features are used, the code and related dependencies for the unused components, like those handling cookies, sessions, or view engines, are still installed onto the developer's machine along with the *express* package.

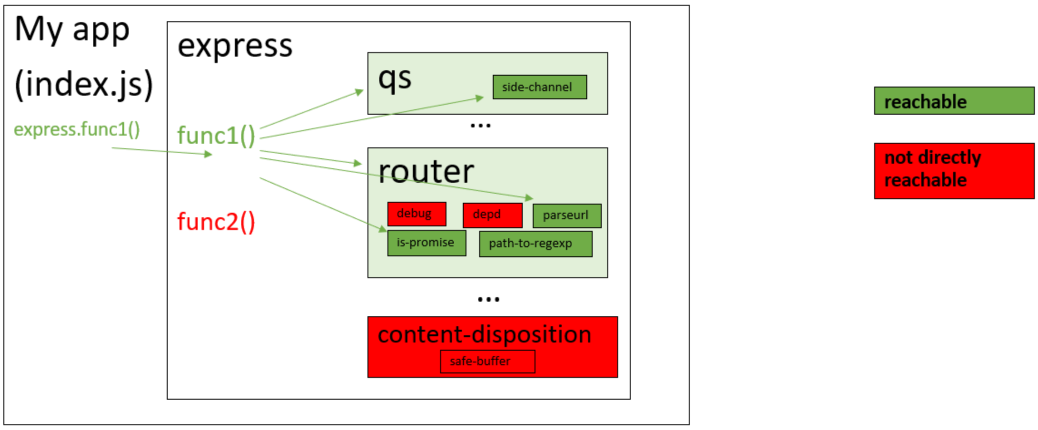

Figure 3: Reachability in packages

For example, the image above illustrates how only specific parts of a dependency graph may be actually reachable at runtime. In this scenario, **My app** (index.js) directly invokes *express.func1()*. This call causes the *qs* module and the router module (along with its internal dependencies like *depd, parseurl* and *path-to-regexp*) to be loaded and become reachable (indicated by the green boxes). Notice that even though router contains other internal modules like debug and depd, these are shown as 'not directly reachable' (red) because *func1()*'s execution flow simply doesn't require them.

The image also shows *func2*(), which is a function within the *express* package that **My app** does not call. Because *func2()* is never invoked, certain modules it might use, such as content-disposition and

([9]) Basic routing endpoints: these are the fundamental paths and handlers (functions) that allow a web application to respond to specific web requests, such as a GET request to retrieve data from the /users path or a POST request to submit a form. They represent the core, essential functionality of a web framework.

its dependency safe-buffer, remain entirely installed but not directly reachable by the application (also indicated by red boxes).

In this context, "not directly reachable" means that the application doesn't explicitly invoke those components. However, it is important to note that this does not guarantee they are entirely isolated. Vulnerable components could still be exposed through unexpected paths (e.g. reflection, misconfiguration, or crafted input). A reachability or exploitability assessment would eventually have to take such possibilities into account.

Ultimately, this distinction is the key to assessing security risk. We must separate **installed code** (the full content on disk) from **reachable code** (the portion active during execution). If a vulnerability exists in a module that is never executed, or in a function that is never invoked, the application is less likely to be exploitable.

2.2 Package manager benefits

Packages and package managers bring significant benefits to software development.

- **Collaboration.** They enable developers to share and reuse modules across projects.
- **Efficiency.** They reduce the need to write functionality from scratch.
- **Consistency.** They promote standardised components and practices.
- **Maintainability.** They simplify updates by centralising dependency management.
- **Quality.** They enable repeated use and testing across many projects, which can improve reliability.

However, this same interconnectedness means that a single vulnerable or compromised package can affect thousands of applications. For example, npm's *express* package currently shows nearly 100 000 packages directly depending on it on the official registry. When factoring in transitive dependencies, the total number of projects relying on *express* can be estimated to be well over a million. This means that a vulnerability in *express* can potentially introduce a security flaw across a vast segment of the software supply chain, impacting hundreds of thousands of individual projects that rely on it.

3. Security risks in package consumption

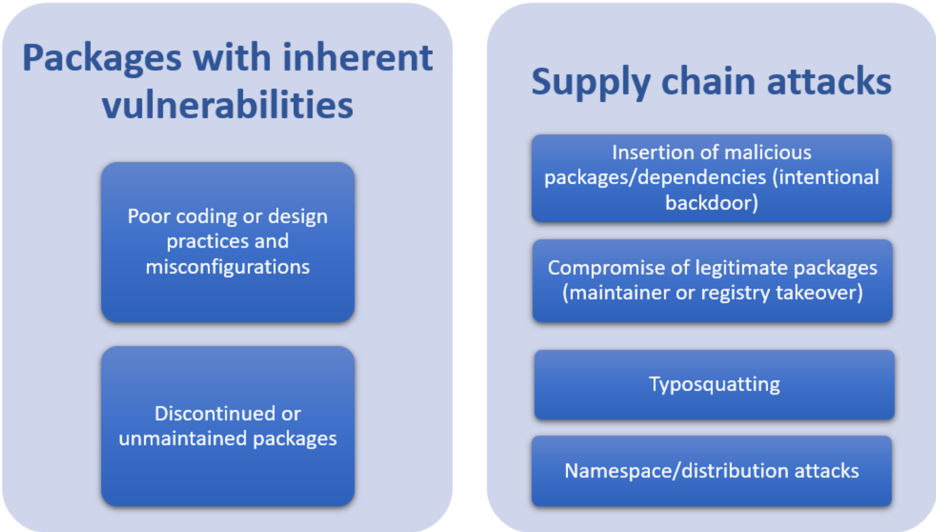

Figure 4: Categories of security risks

This section examines the security risks that developers face when consuming third-party packages via package managers. Note that this analysis specifically excludes scenarios involving a compromise of the package distribution repository itself. It also excludes specific vulnerabilities that result from poor coding practices.

While package managers provide several advantages, they also introduce security risks that can affect both developers and downstream users. To better understand these risks, we group them into two main categories:

- **packages with inherent vulnerabilities,** and
- **supply chain attacks**, as shown in the figure above.

The first category focuses on weaknesses within the packages themselves, whereas the second category covers threats arising from the software distribution process.

3.1 Packages with inherent vulnerabilities

Packages can contain vulnerabilities that result from flaws in their own code, design or configuration. These vulnerabilities may be introduced unintentionally by maintainers or intentionally by malicious actors. Developers making use of third-party packages must recognise that a vulnerability in an imported package can compromise the security of their entire application.

3.1.1 Poor coding or design practices and misconfigurations

Packages may include security flaws resulting from unsafe coding or design practices, such as weak input validation or insecure cryptographic implementations. Misconfigurations, like overly permissive defaults or exposed debug modes, can also increase the attack surface.

This document does not attempt to enumerate all vulnerability types introduced by insecure coding practices within third-party packages. Many such weaknesses fall under well-known CWE classes, such as improper input validation (CWE-20) ([10]), improper limitation of pathname (CWE-22) ([11]), sensitive information exposure (CWE-200) ([12]) and deserialisation of untrusted data (CWE-502).

When developers consume third-party packages, these vulnerabilities become part of their application's attack surface, even if the developer did not introduce them directly.

For example, the latest available node-serialize package ([13]) contains a known critical deserialisation of untrusted data (CWE-502) vulnerability ([14]), with the recommended remediation being to simply not use this package. This issue is compounded by the fact that the project has been discontinued, which leads to the next risk.

3.1.2 Discontinued or unmaintained packages

Unmaintained or abandoned packages, sometimes referred to as open-source abandonware, can pose significant security risks. Without active maintainers, vulnerabilities remain unpatched and dependencies become outdated.

The recent *TARmageddon* incident ([15]), which stemmed from the abandoned Rust tokio-tar library ([16]), illustrates how unmaintained packages can resurface as attack vectors even several years later. Similarly, the previously mentioned *node-serialize* package remains available on the npm registry in its vulnerable version (0.0.4 – released 11 years ago) and continues to receive weekly downloads. Even widely used projects are not immune to this. For example, the popular crypto-js library ([17]), although not currently affected by known vulnerabilities, is listed as discontinued on its repository page, yet it continues to record millions of downloads each week.

([10]) NIST, National Vulnerability Database, CVE-2021-22931 Detail, https://nvd.nist.gov/vuln/detail/cve-2021-22931, accessed 28 November 2025.
([11]) NIST, National Vulnerability Database, CVE-2025-27210 Detail, https://nvd.nist.gov/vuln/detail/CVE-2025-27210, accessed 28 November 2025.
([12]) NIST, National Vulnerability Database, CVE-2023-45143 Detail, https://nvd.nist.gov/vuln/detail/cve-2023-45143, accessed 28 November 2025.
([13]) NPM, 'node-serialize', n.d., https://www.npmjs.com/package/node-serialize, accessed 28 November 2025.
([14]) ENISA, EU Vulnerability Database, EUVD-2018-0297, 8 May 2024, https://euvd.enisa.europa.eu/vulnerability/CVE-2017-5954, accessed 28 November 2025.
([15]) Edera, 'TARmageddon (CVE-2025-62518): RCE Vulnerability Highlights the Challenges of Open Source Abandonware', 21 October 2025, https://edera.dev/stories/tarmageddon, accessed 28 November 2025.
([16]) ENISA, EU Vulnerability Database, EUVD-2025-35176, 16 January 2026, https://euvd.enisa.europa.eu/vulnerability/CVE-2025-62518, accessed 28 November 2025.
([17]) NPM, 'crypto-js', n.d., https://www.npmjs.com/package/crypto-js, accessed 28 November 2025.

3.2 Supply chain attacks

A supply chain refers to the ecosystem of processes, people, organisations and distributors involved in the creation and delivery of a final solution or product ([18]). The supply chain can be understood in terms of four key elements:

- **suppliers**, for example package maintainers;
- **supplier assets**, for example source code, build pipelines, publishing credentials and released packages;
- **customers**, for example application developers who consume the third-party packages;
- **Customer assets**: e.g., the consuming application, its environment, data, and end-users

A supply chain attack typically consists of two linked stages: an attacker compromises a supplier or its assets, and then leverages that compromised supplier to attack the customer ([19]). In this model, both the supplier and the customer become targets.

This model directly applies to the package ecosystems. Attackers can attack maintainer accounts, inject malicious code into a package's build pipeline or manipulate distribution channels, all of which represent attacks on the supplier. Any developer consuming these affected packages becomes a downstream customer, inheriting the compromise at the customer level.

Supply chain attacks often exploit the interconnected nature of the package ecosystem, often targeting its weaker links, i.e. areas where defences may be limited because of their perceived insignificance in isolation, such as an obscure library that only performs only small remedial tasks within a complex software solution. An advantage for attackers is that compromising a single link can affect every application or developer who depends on that link within their chain, allowing the impact to cascade across multiple systems and, ultimately, end users. The following subsections describe common categories of supply chain attacks observed in package ecosystems.

3.2.1 Insertion of malicious packages/dependencies

Attackers can publish entirely new packages containing malicious code with the goal of having application or package developers include their malicious packages as dependencies. When these packages are installed directly or through transitive dependencies, the malicious code executes within these otherwise legitimate applications. As a result, the legitimate applications could end up exfiltrating data and credentials ([20]), or performing other malicious activities like cryptocurrency mining.

An example of an inherently malicious package is *crypto-encrypt-ts* ([21]). The npm registry has since flagged this package as malicious but unfortunately, detection is not always immediate, and some malicious packages may remain available for a long time before being discovered.

Malicious package insertion is not limited to traditional software dependencies. Malicious developer extensions, plugins and tooling packages can also serve as vectors to spread malware across

([18]) Beamon, B. M. (1998), 'Supply chain design and analysis: Models and methods', *International Journal of Production Economics*, Vol. 55, Issue 3, pp. 281–294, https://doi.org/10.1016/S0925-5273(98)00079-6.
([19]) ENISA, *ENISA threat landscape for supply chain attacks*, 2021, https://data.europa.eu/doi/10.2824/168593, accessed 11 February 2025.
([20]) Sonatype, 'Open Source Malware Index Q2 2025: Data Exfiltration Remains a Leading Threat', 8 July 2025, https://www.sonatype.com/blog/open-source-malware-index-q2-2025, accessed 28 November 2025.
([21]) Hack Read, 'npm Malware Targets Crypto Wallets, MongoDB; Code Points to Turkey', 1 May 2025, https://hackread.com/npm-malware-crypto-wallets-mongodb-turkey-code/, accessed 28 November 2025.

development environments. For example, several Visual Studio Code extensions were recently found to contain malware (GlassWorm), directly compromising developer machines through their IDE ([22]). This approach follows a similar supply-chain distribution approach through the extension package marketplace instead of package repositories.

3.2.2 Compromised legitimate packages

Even well-established packages can be hijacked through malicious or compromised maintainer accounts, stolen credentials or social engineering. If an attacker manages to gain publishing rights, they can insert malicious code into one or more legitimate packages, which are then proliferated onto downstream projects that depend on these packages.

The *event-stream* incident ([23]) was a notable case in which a malicious package (*flatmap-stream*) was added by a new maintainer to a widely used npm package, injecting malicious code that targeted a specific cryptocurrency wallet application.

In a more recent incident, multiple legitimate packages were compromised to deliver credential-stealing malware ([24]). Instead of introducing the packages as a new maintainer, the attackers gained access to existing maintainers' accounts through social engineering tactics, and proceeded to publish modified and malicious versions of popular packages.

Similarly, in another incident, attackers gained control of the maintainer's account of the widely used *ua-parser-js* packages. The attackers proceeded to add malicious code to the pre-install and post-install scripts, enabling automatic execution of the payload during installation ([25]).

A compromise can also be introduced into previously legitimate and widely used packages. For example, in 2022 the maintainer of the *colors* and *faker* npm packages included intentionally disruptive code ([26]) that caused dependent applications to fail.

3.2.3 Typosquatting

Package distribution attacks manipulate the way packages are retrieved or resolved by package managers. By exploiting weaknesses in naming, versioning or repository trust, attackers can force victim developers to download unintended or malicious package versions.

Typosquatting is a simple but effective package distribution attack, whereby an attacker publishes malicious packages with names similar to legitimate ones, with the hope that developers will install them by mistake. Examples such as the malicious *crossenv* package ([27]), which purposely chose a

([22]) The Hacker News, 'GlassWorm Malware Discovered in Three VS Code Extensions with Thousands of Installs', 10 November 2025, https://thehackernews.com/2025/11/glassworm-malware-discovered-in-three.html, accessed 28 November 2025.
([23]) NPM, 'Details about the event-stream incident', 27 November 2018, https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident, accessed 28 November 2025.

([24]) Palo Alto Networks, 'Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk', 10 September 2025, https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/, accessed 28 November 2025.
([25]) Sonatype, 'Popular npm Project Used by Millions Hijacked in Supply-Chain Attack', 25 October 2021, https://www.sonatype.com/blog/npm-project-used-by-millions-hijacked-in-supply-chain-attack, accessed 28 November 2025.
([26]) Snyk, 'Open source maintainer pulls the plug on npm packages colors and faker, now what?', 9 January 2022, https://snyk.io/fr/blog/open-source-npm-packages-colors-faker/, accessed 11 February 2026.

([27]) NPM, 'crossenv', n.d., https://www.npmjs.com/package/crossenv, accessed 28 November 2025.

name similar to the popular legitimate *cross-env* package ([28]), show how a single mistyped letter can lead to unintentional inclusion of malicious code through misleading package instances ([29]).

3.2.4 Namespace/Dependency Confusion

Namespace ([30]) or dependency ([31]) confusion is another package distribution attack that exploits differences between internal and public repositories. In environments where both internal and public registries are configured, package managers may automatically resolve dependencies from either source based on the name and version. Attackers publish packages with the same names as private packages with a much higher version number, thus tricking package managers into fetching the malicious public version instead of the private package. The attack succeeds as a result of permissive version ranges and lack of internal repository enforcement. This attack vector was demonstrated in practice in 2021, showing how dependency confusion attacks can lead to unintended installation of externally published packages ([32]).

3.3 Consequences

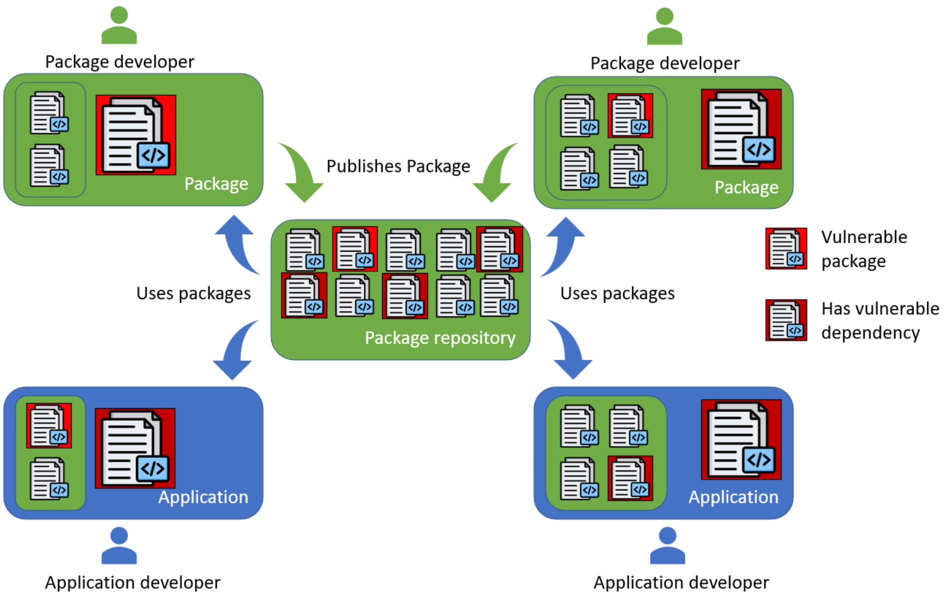

Figure 5: Propagation of vulnerabilities

In both cases of inherent vulnerabilities or supply chain attacks, the advantages of package managers, that allow for code reuse and easy integration, can also become their greatest weakness.

As shown in the figure above, a single vulnerable or compromised package can propagate through thousands or even millions of projects and environments. What enables speed and collaboration also enables the proliferation of vulnerabilities, where one malicious dependency can rapidly cascade

([28]) NPM, 'cross-env', n.d., https://www.npmjs.com/package/cross-env, accessed 28 November 2025.
([29]) NPM, "'crossenv' malware on the npm registry", 2 August 2017, https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry, accessed 28 November 2025.
([30]) Sonatype, 'Namespace Confusion Protection', 29 January 2025, https://help.sonatype.com/en/namespace-confusion-protection.html, accessed 28 November 2025.
([31]) Blaze, 'Dependency Confusion: An exploitation overview', 26 February 2023, https://www.blazeinfosec.com/post/dependency-confusion-exploitation/, accessed 28 November 2025.
([32]) Medium, 'Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies', 9 February 2021, https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610, accessed 1 February 2026.

across entire software ecosystems. For example, the recent npm supply chain attack, targeting 18 widely used packages with a combined download volume of over 2.6 billion downloads per week ([33]), demonstrated how a single coordinated attack can impact a large portion of the Node.JS ecosystem.

Similarly, the recent critical React vulnerability ([34]) (CVE-2025-55182), with a CVSS score of 10.0, demonstrates how vulnerabilities in highly adopted frameworks can create an exceptionally large blast radius. Early analysis estimated that over 12 million websites may be affected ([35]), highlighting the scale at which a vulnerability in a widely adopted dependency can propagate.

([33]) Palo Alto Networks, 'Breakdown: Widespread npm Supply Chain Attack Puts Billions of Weekly Downloads at Risk', 10 September 2025, https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/, accessed 28 November 2025.

([34]) React, 'Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies, 3 December 2025, https://react.dev/blog/2025/12/03/critical-security-vulnerability-in-react-server-components, accessed 11 December 2025.

([35]) HackerOne, 'CVE-2025-55182: Unauthenticated React Exploit Affects Millions of Sites', 5 December 2025, https://www.hackerone.com/blog/cve-2025-55182-react-exploit, accessed 11 December 2025.

4. Best practices – secure package consumption

This section describes best practices that developers can follow when consuming third-party packages. For simplification, we describe the package consumption life cycle, as shown in the figure below, in which a developer selects suitable packages, integrates them into the project and subsequently monitors and mitigates any identified issues.

The risks and threats outlined in the previous section, along with the best practices described in this section, align with the stages of the package consumption life cycle.

Stage	Threats	Security objectives
Select	Selecting packages with inherent vulnerabilities (3.1)	Choosing trustworthy, verified, and well-maintained packages
	Selecting malicious packages distributed through supply chain attacks (3.2)	
Integrate	Integrating packages with inherent vulnerabilities (3.1)	Ensuring secure integration of selected packages
	Integrating malicious packages distributed through supply chain attacks (3.2)	
Monitor	Integrated packages contain newly detected vulnerabilities (3.1)	Detecting vulnerabilities or compromised packages
	Integrated legitimate packages are compromised (3.2)	
Mitigate	Integrated packages contain newly detected vulnerabilities (3.1)	Applying patches, or isolating, or removing affected packages in a timely manner to maintain a secure state
	Integrated legitimate packages are compromised (3.2)	

In the first two stages (package selection and integration), developers should aim to embed secure-by-design principles and controls, ensuring that security is considered from the outset when incorporating packages into a project. In the latter stages (monitoring and mitigation), developers should aim to implement security controls that sustain and enforce the security posture over time.

The following subsections provide best practice recommendations for each stage of the package consumption life cycle. This is meant to serve as practical guidance that begins with selecting trustworthy packages, continues through secure integration and extends into continuous monitoring and timely mitigation.

NB: The tools and commands included in this section are illustrative examples only and do not represent a recommendation of specific tools. Their suitability, applicability, and impact

depend on the specific environment and should be assessed, tailored, and managed by the implementor.

4.1 Package selection

This stage focuses on evaluating and choosing packages that demonstrate trustworthiness, active maintenance and transparent security practices before inclusion in a project. It is important to note that before evaluating the trustworthiness of a dependency, developers should first assess whether the dependency is actually needed. Reducing the number and scope of dependencies can lower the attack surface and simplify the eventual security management.

4.1.1 Selection recommendations

The following table lists recommendations that can be used to support the security objective of choosing trustworthy, verified and well-maintained packages. Each recommendation represents a specific aspect of package trustworthiness, helping developers evaluate integrity, reputation, maintenance activity and transparency before adoption

Selection recommendation	Description	Related threats
Trusted source	Use only official and verifiable package registries, and prefer packages published through secure workflows such as Trusted Publishing, which provide provenance metadata to verify the publisher's identity.	3.2
Existing known vulnerabilities	Check vulnerability databases and scanning tools, such as npm audit, Snyk and OSV, for existing issues before use.	3.1
Package signing and integrity verification	Use packages and repositories that provide cryptographic signing to verify integrity.	3.2.2, 3.2.4
Maintainer reputation	Select packages maintained by reputable or verified organisations/publishers with a consistent record.	3.1, 3.2.1, 3.2.3
Popularity and maintenance	Consider community adoption and recent update activity as indicators of reliability and ongoing support. Examples include project stars, downloads and commits. NB: Popularity metrics can be misleading or artificially inflated ([36]) and should not be relied upon in isolation.	3.1, 3.2.1, 3.2.3
Usage of secure practices	Evaluate package build practices to ensure they avoid unnecessary or insecure installation behaviours.	3.1, 3.2

4.1.2 Cheat Sheet

The following table complements the selection recommendations by providing practical examples, tools and approaches for package selection. It outlines examples of checks, mainly focusing on npm and GitHub, that can help developers verify provenance, maintainer reputation, update activity and overall security posture when selecting packages. This list is not exhaustive but is intended to serve as a practical starting point for practical secure package selection.

([36]) He, H., Yang, H., Burckhardt, P., Kapravelos, A., Vasilescu, B. and Kästner, C. (2026), 'Six Million (Suspected) Fake Stars on GitHub: A Growing Spiral of Popularity Contests, Spam, and Malware', Proceedings of the 48th International Conference on Software Engineering (ICSE 2026), Rio de Janeiro, Brazil, 12–18 April 2026.

Selection recommendation	Examples

Trusted source	Several package managers allow developers to publish using provenance statements ([37]) to prove that a package was built by a trusted publisher ([38]).
	• Verify the package's provenance to confirm where and how it was published, and that it was released by a trusted and authorised publisher ([39]):
	  ○ look for a green check mark on the dedicated npm package web page;
	  ○ *npm audit signatures*.
	• Prefer packages that include valid provenance metadata.
	• Verify package names carefully to avoid malicious imitations or naming collisions.

Existing known vulnerabilities	• Run scans prior to installation or during dependency review:
	  ○ *npm audit --json*;
	  ○ *osv-scanner -r.*;
	  ○ *dependency-check --project "My Project" --scan./* ([40]).
	• Consult public databases such as EUVD ([41]), OSV.dev ([42]), Snyk ([43]) or NVD ([44]) to check the package's vulnerability history.
	• Avoid or postpone adoption of packages with unresolved high/critical severity issues.

Package signing and integrity verification	• Check that the ecosystem supports integrity metadata, such as SHA-512 hashes in npm (package-lock.json) or enforceable hashes in pip (*pip install --require-hashes*).
	• Prefer package managers and packages that support provenance verification.
	• Avoid packages that bypass registry verification, such as direct installs from GitHub or tarball URLs, which lack provenance and integrity checks.
	• Verify that packages include provenance metadata:
	  ○ *npm audit signatures* reports 'missing provenance'.
	• Avoid packages that use pre-install or post-install scripts to download additional code or binaries from external URLs:
	  ○ *npm view <package> scripts*;
	  ○ *grep -R 'curl|wget|git clone" node_modules/*/package.json*.
	• *grep -R 'curl|wget|git clone" node_modules/*.

Maintainer reputation	• Review maintainer metadata, for example:
	  ○ *npm view <package> contributors*;
	  ○ *npm view <package> maintainers*;
	  ○ check npm and/or code repository (e.g. github/gitlab) webpage info;
	  ○ *npm info express -json* (maintainers, author, contributors, publisher – most recent publisher).
	• Be cautious of packages owned by newly created, single project accounts with no additional contributors:
	  ○ *https://registry.npmjs.org/-/v1/search?text=maintainer:<enter maintainer ID here>*.
	• Prefer packages from verified publishers (Github/npm) and those that include valid provenance metadata.
	• Where feasible, consider maintaining an internal 'allowlist' of approved packages or maintainers, potentially informed by trusted industry stakeholders or open-source foundations.

([37]) NPM, 'Generating provenance statements', 4 August 2025, https://docs.npmjs.com/generating-provenance-statements, accessed 28 November 2025.
([38]) NPM, 'Trusted publishing for npm packages', 10 February 2026, https://docs.npmjs.com/trusted-publishers, accessed 28 November 2025.
([39]) NPM, 'Viewing package provenance', 7 July 2025, https://docs.npmjs.com/viewing-package-provenance, accessed 28 November 2025.
([40]) OWASP Dependency-check supports Node.js but is primarily designed for Maven/Java projects.
([41]) ENISA, 'EU Vulnerability Database', https://euvd.enisa.europa.eu/, accessed 28 November 2025.
([42]) OSV, 'Vulnerabilities', https://osv.dev/list?ecosystem=npm, accessed 28 November 2025.
([43]) Snyk, 'Vulnerabilities', https://security.snyk.io/vuln/npm, accessed 28 November 2025.
([44]) NIST, 'National Vulnerability Database', https://nvd.nist.gov/, accessed 28 November 2025.

Popularity and maintenance	• Check for recent community engagement: ○ check github insights tab (contributors, commits, etc.); ○ check CHANGELOG.md and SECURITY.md on repository webpage; ○ check open or recently closed issues and pull requests on repository webpage; ○ *curl -s 'https://api.github.com/repos/<owner>/<repo>/issues?state=all&per_page=100" | jq '.[] |.state,.created_at,.closed_at'*. • Prefer active projects with frequent commits and tagged releases, for example: ○ *git shortlog -sne*; ○ *npm info <package> -json* (time, versions). • Review documentation and security contact details to confirm openness and accountability. • Prefer dependencies maintained by verified and known active organisations or foundations.
Usage of secure practices	• Inspect the package's life cycle scripts for potentially unsafe commands such as pre-install or post-install: ○ *npm view <package> scripts*. • Review dependency trees to confirm the package introduces only necessary dependencies and no excessive or nested components: ○ npm ls --all.

4.2 Package integration

This stage focuses on securely integrating the selected packages into the development and build process. The goal is to maintain package integrity, provenance, and source trust throughout the software build life cycle.

4.2.1 Integration recommendations

The following table lists integration recommendations that support secure and verifiable package use. These recommendations focus on integrity, source verification and automation, ensuring that once packages are selected, they are installed and managed securely.

Integration recommendation	Description	Related threats
SBOM creation	Generate a software bill of materials (SBOM) to document all dependencies for future security and compliance tracking.	3.1
Vulnerability checks	Enforce security policies in CI/CD pipelines to prevent builds from proceeding with known vulnerable components.	3.1
Integrity enforcement	Enforce hash or lockfile verification to confirm that installed packages match approved versions.	3.2.2, 3.2.4
Package source enforcement/verification	Restrict installations to trusted package registries and validate source URLs.	3.2.1, 3.2.3, 3.2.4
Installation script prevention	Inspect and disable or restrict scripts executed during installation to reduce attack surface. NB: Disabling scripts may impact packages and functionality. This control may be more suited for high-security or isolated environments.	3.1, 3.2
Pinning versions	Fix dependency versions to prevent unverified updates from introducing new vulnerabilities.	3.1, 3.2.2, 3.2.4

4.2.2 Cheat sheet

The following table complements the integration recommendations by providing practical examples and commands for their implementation. This list is not exhaustive and examples are primarily drawn from npm and Node.js, but equivalent approaches apply across other package manager ecosystems.

Integration recommendation	Examples
SBOM creation	Generate an SBOM during build: • *syft. -o spdx-json > sbom.json* ([45]); • *@cyclonedx/cyclonedx-npm --output-file sbom.json* ([46]).
Vulnerability checks	Block build/install when vulnerabilities are found: • *npm audit --omit=dev --audit-level=high* ([47]); • *grype sbom:./sbom.json --fail-on High* ([48]).
Integrity enforcement	npm automatically enforces integrity by verifying SHA-512 hashes stored in package-lockfile.json. Other ecosystems may explicitly require enforcement, for example with pip: • *pip install --require-hashes -r requirements.txt* ([49]).
Package source enforcement/verification	Use a local package registry or proxy (e.g. Verdaccio, Nexus Repository, JFrog Artifactory, AWS CodeArtifact, GitHub Packages or Google Artifact Registry) to ensure packages are sourced only from approved and verified repositories. Configure.npmrc file to restrict installations to the trusted registry: • *echo ‘registry=https://<registry>“ >>.npmrc*. NB: A local package registry or proxy with caching capabilities can also support business continuity by retaining previously retrieved packages, even if they are removed from the upstream registry or if the registry itself is temporarily unavailable.

Installation script prevention	Disable the execution of installation scripts from third-party packages:
	• run installs with scripts disabled:
	  ○ *npm install --ignore-scripts <package>* ([50]).
	• set this as a project or global policy:
	  ○ *echo ‘ignore-scripts=true’ >> .npmrc*;
	  ○ *npm config set ignore-scripts true -g*.
	NB: Disabling scripts may impact packages and functionality.
	This control may be more suited for high-security or isolated environments.

Pinning versions	• Use a lockfile (package-lock.json ([51]), yarn.lock, etc.) to maintain exact dependency versions. • Commit both package.json and the lockfile to source control. • Enforce lockfile usage in CI/CD ([52]): ○ *npm ci*. • Review changelogs and release notes before upgrading dependencies.

(45) Github, ‘anchore/syft’, https://github.com/anchore/syft, accessed 28 November 2025.
(46) Github, ‘CycloneDX/cyclonedx-node-npm’, https://github.com/CycloneDX/cyclonedx-node-npm, accessed 28 November 2025.
(47) NPM, ‘npm-audit’, 26 October 2022, https://docs.npmjs.com/cli/v8/commands/npm-audit, accessed 28 November 2025.
(48) Github, ‘anchore/grype’, n.d., https://github.com/anchore/grype, accessed 28 November 2025.
(49) PyPA, ‘Secure installs’, n.d., https://pip.pypa.io/en/stable/topics/secure-installs/, accessed 28 November 2025.
(50) OWASP Cheat Sheet Series, ‘NPM Security best practices’, n.d., https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html, accessed 28 November 2025.
(51) NPM, ‘package.json’, 26 October 2022, https://docs.npmjs.com/cli/v8/configuring-npm/package-json, accessed 28 November 2025.
(52) OWASP Cheat Sheet Series, ‘NPM Security best practices’, n.d., https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html, accessed 28 November 2025.

	• Optional and situational: install packages at a specific date to avoid newly published versions: ○ *npm install express --before="$(date -d '-7 days')'.* **NB: Pinning versions can prolong exposure to known vulnerabilities if updates are not regularly reviewed and applied.**

4.3 Package monitoring

This stage focuses on maintaining visibility over the security posture of integrated packages. Monitoring ensures that new vulnerabilities, deprecations, or maintainer changes are detected early, allowing for timely remediation before they can be exploited.

4.3.1 Package monitoring recommendations

The following table outlines controls for maintaining visibility and control over package security after integration.

Package monitoring recommendation	Description	Related threats
SBOM-driven monitoring	Leverage SBOM data to automate vulnerability correlation and streamline monitoring workflows.	3.1, 3.2.1, 3.2.2
Automate vulnerability scanning in CI/CD	Continuously scan dependencies to identify new or resurfacing vulnerabilities.	3.1, 3.2.1, 3.2.2
Track CVEs/advisories	Monitor vulnerability databases regularly to stay informed about newly reported issues.	3.1, 3.2.1, 3.2.2
Monitor for outdated version release	Monitor existing packages for the availability of newer versions.	3.1, 3.2.1, 3.2.2
Set alerts	Set alerts on specific events deemed risky for your organisation, for example: • on new CVEs affecting locked versions; • on deprecated or unmaintained releases; • on lack of maintainer activity. • on maintainer ownership change;	3.1, 3.2.1, 3.2.2

4.3.2 Cheat sheet

The following table complements the monitoring controls by providing practical examples and tools for their implementation. The list is not exhaustive, and examples are primarily drawn from npm and Node.js, though equivalent practices apply across other package management ecosystems.

Package monitoring recommendation	Examples
SBOM-driven monitoring	Use SBOM data for periodic/continuous vulnerability checks: • *grype sbom:./sbom.json* ([53]); • *osv-scanner scan --sbom.json* ([54]).

([53]) Github, 'anchore/grype', n.d., https://github.com/anchore/grype, accessed 28 November 2025.
([54]) Github, 'google/osv-scanner', n.d., https://github.com/google/osv-scanner, accessed 28 November 2025.

Automate vulnerability scanning in CI/CD	Integrate dependency scanning commands into pipeline stages: • *npm audit --omit=dev --audit-level=high*; • *osv-scanner -r.* Schedule periodic scans and fail builds if new high-severity vulnerabilities are detected.
Track CVEs/advisories	Subscribe to and monitor public vulnerability feeds: • EUVD ([55]), OSV.dev ([56]), Snyk ([57]) or NVD ([58]); • use GitHub Dependabot ([59]) or npm audit ([60]) to monitor and notify of new issues affecting dependencies; • vendor-published machine-readable advisories, such as the Common Security Advisory Framework (CSAF) ([61]), could support automated ingestion and processing.
Monitor outdated version release	Monitor offset between installed and available versions: • *npm outdated*.

Set alerts	Define items to monitor for alerts, for example:
	• new CVEs affecting locked versions:
	  ○ GitHub Dependabot alerts ([62]);
	  ○ *grype sbom:./sbom.json*;
	  ○ *osv-scanner scan --sbom.json*.
	• deprecated or unmaintained releases:
	  ○ *npm view <package> deprecated*;
	  ○ *npm info <package> time.modified*;
	• lack of maintainer activity
	  ○ certain commands mentioned in section 4.1.2 can be performed over time to monitor the maintainer activity (or lack thereof) on a specific package
	• maintainer or ownership changes:
	  ○ *npm view <package> maintainers --json*.
	Configure alerts using internal notification process (e.g. email, Slack or Teams) and integrate alerting into CI/CD.

4.4 Vulnerability mitigation

This stage focuses on assessing, prioritising and addressing vulnerabilities detected in third-party packages. The detection of a vulnerability in a dependency typically means that under specific conditions, the package contains code that could be exploited. However, this does not necessarily mean that your application is vulnerable as a result of this vulnerability.

Many software composition analyses or static analysis tools tend to report vulnerabilities based on the presence of a specific version of a library that has a known CVE. While these findings indicate a potential risk, they may lack contextual awareness, for example whether the affected function described in the CVE is actually imported, reachable or executed in your codebase.

([55]) ENISA, 'EU Vulnerability Database', https://euvd.enisa.europa.eu/, accessed 28 November 2025.
([56]) OSV, 'Vulnerabilities', https://osv.dev/list?ecosystem=npm, accessed 28 November 2025.
([57]) Snyk, 'Vulnerabilities', https://security.snyk.io/vuln/npm, accessed 28 November 2025.
([58]) NIST, 'National Vulnerability Database', https://nvd.nist.gov/, accessed 28 November 2025.
([59]) Github, 'About Dependabot alerts', n.d., https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts, accessed 28 November 2025.
([60]) NPM, 'npm-audit', 23 October 2023, https://docs.npmjs.com/cli/v9/commands/npm-audit, accessed 28 November 2025.
([61]) CSAF, 'Common Security Advisory Framework (CSAF)', n.d., https://www.csaf.io/, accessed 1 February 2026.
([62]) Github, 'About Dependabot alerts', n.d., https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts, accessed 28 November 2025.

Effective vulnerability mitigation involves assessing findings to determine their relevance and exploitability before deciding on a prioritisation and remediation.

4.4.1 Vulnerability mitigation steps

The following table outlines the steps a developer can take to mitigate a vulnerability after detection.

Vulnerability mitigation recommendation	Description	Related threats
Assess	Assess reported vulnerabilities, retrieving data and metrics relating to their exploitability, relevance and reachability within the current system context.	3.1, 3.2.1, 3.2.2
Prioritise	Define risk appetite and prioritise vulnerabilities based on severity, exploitability and potential impact.	3.1, 3.2.1, 3.2.2
Mitigate	Apply the appropriate remediation action such as upgrading, mitigating/isolating or rolling back.	3.1, 3.2.1, 3.2.2
Document and notify	Record actions taken, update the SBOM and communicate changes as part of regulatory or organisational compliance obligations.	3.1, 3.2.1, 3.2.2

4.4.2 Cheat sheet

The following table complements the vulnerability mitigation steps by providing practical examples and tools for their implementation. The list is not exhaustive, but rather serves to provide some practical examples of processes and tools that a developer could use.

Vulnerability mitigation recommendation	Examples
Assess	For identified vulnerabilities: • Retrieve and record vulnerability metrics (CVSS, EPSS, KEV, VEX): ○ CVSS: Severity score and vector (e.g., OSV.dev, NVD, Snyk or EUVD); ○ EPSS (63): Probability of exploitation score; ○ KEV (64): Check if CVE is listed in CISA's Known Exploited Vulnerabilities catalogue; ○ VEX (65) data: Vulnerability Exploitability eXchange format used by software suppliers to describe whether vulnerabilities affect a product. • Retrieve and assess vendor-provided advisory information. Machine-readable formats (e.g. CSAF) could help to automate vulnerability information processing. • Perform reachability/context analysis (e.g. CodeQL (66), Semgrep (67), call graph/static analysis tools or other commercial tools like snyk (68) code) to confirm if vulnerable code is actually used.
Prioritise	• Define risk thresholds (e.g. 'treat CVSS ≥ 7.0 or EPSS ≥ 0.3 as high priority"). • Focus remediation on exploitable and reachable vulnerabilities with high severity or exploitation scores above risk threshold. • Prioritise vulnerabilities affecting dependencies in production environments.

(63) FIRST, 'EPSS Data', n.d., https://www.first.org/epss/data_stats, accessed 28 November 2025.
(64) CISA, 'Known Exploited Vulnerabilities Catalog', https://www.cisa.gov/known-exploited-vulnerabilities-catalog, accessed 28 November 2025.
(65) CycloneDX, 'Vulnerability Exploitability eXchange (VEX)', n.d., https://cyclonedx.org/capabilities/vex, accessed 28 November 2025.
(66) Github, 'CodeQL', n.d., https://codeql.github.com/, accessed 28 November 2025.
(67) Github, 'semgrep', n.d., https://github.com/semgrep/semgrep, accessed 28 November 2025.
(68) Snyk, 'Reachability analysis', n.d., https://docs.snyk.io/manage-risk/prioritize-issues-for-fixing/reachability-analysis, accessed 28 November 2025.

Mitigate	• Patch or upgrade:
	    ○ *npm update <package>*;
	    ○ *npm install <package>@<version>*.
	• Use temporary mitigating controls (e.g. WAF rules, feature flags) to limit exposure until a fix is applied.
	• Remove, roll back or isolate if no patch is available.

Document and notify	• Categorise vulnerabilities as exploitable, non-exploitable or mitigated by configuration:
	    ○ VEX Product Statuses: *known_affected, known_not_affected, under_investigation, and fixed*.
	• Generate/update a VEX mitigation statement.
	• Update SBOM:
	    ○ *syft. -o spdx-json > sbom.json*;
	    ○ *@cyclonedx/cyclonedx-npm --output-file sbom.json*.
	• Update release notes.
	• Notify relevant internal/external stakeholders (e.g. impacted customers, relevant authorities, DevSecOps, risk management, compliance teams).

5. Future considerations

This document is intended to serve as a starting point with concise guidance on package consumption within software projects, encouraging risk aware decision making when consuming and managing third-party packages. While many of the listed examples focus on npm, pip and GitHub, the recommendations are designed to apply across package manager ecosystems.

Existing resources, such as OpenSSF's npm Best Practices for the Supply Chain ([69]) and the OWASP NPM Security Cheat Sheet ([70]), offer ecosystem-specific detailed implementation guidance. Organisations and developers are encouraged to consult and align with existing guidance and best practices in addition to this document. Activities such as third-party package selection can further benefit from related guidance, including resources from well-established industry organisations such as the OpenSSF Concise Guide for Evaluating Open Source Software ([71]) and the CISA Software Acquisition Guide ([72]).

The software supply chain landscape continues to evolve, with new tools, processes and risks emerging over time. Therefore, organisations should treat this subject as an ongoing activity and periodically review and update their practices to reflect changes in available tooling, threats and ecosystem-specific guidance. The following subsections provide insight into such evolutions and highlight opportunities like automation and risks like the ones introduced through AI-assisted development.

5.1 Automation

As projects grow in complexity, with hundreds of dependencies across multiple repositories, manual security practices do not scale. Automation, through CI/CD pipelines and dedicated tooling, becomes important to maintain a scalable and secure development workflow.

Most modern CI/CD platforms (e.g. GitHub Actions, GitLab CI, Jenkins) support integration of open-source tools, including:

- Syft or CycloneDX CLI for SBOM generation;
- Grype or OSV-Scanner for vulnerability scanning;
- native package manager commands such as *npm ci --ignore-scripts* for dependency integrity and install-script control.

Open-source tools like Syft and Grype have public examples illustrating how to integrate SBOM generation and vulnerability scanning into CI/CD workflows ([73]). Similarly, other open-source and

([69]) OpenSSF, 'npm Best Practices for the Supply-Chain', 1 September 2022, https://openssf.org/blog/2022/09/01/npm-best-practices-for-the-supply-chain/, accessed 1 February 2026.
([70]) OWASP Cheat Sheet Series, 'NPM Security best practices', n.d., https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html, accessed 1 February 2026.
([71]) OpenSSF, 'Concise Guide for Evaluating Open Source Software', 28 March 2025, https://best.openssf.org/Concise-Guide-for-Evaluating-Open-Source-Software.html, accessed 1 February 2026.
([72]) CISA, 'Software Acquisition Guide for Government Enterprise Consumers: Software Assurance in the Cyber-Supply Chain Risk Management (C-SCRM) Lifecycle', 1 August 2024, https://www.cisa.gov/resources-tools/resources/software-acquisition-guide-government-enterprise-consumers-software-assurance-cyber-supply-chain, accessed 1 February 2026.
([73]) Anchore, 'Generating SBOMs for JavaScript Projects: A Developer's Guide', 28 March 2025, https://anchore.com/blog/javascript-sbom-generation/, accessed 28 November 2025.

commercial solutions often include automation features that help enforce these controls at different stages of the software life cycle.

Moreover, initiatives like the CSAF ([74]) promote the publication of machine-readable vulnerability advisories. The adoption of machine-readable advisories further enables automation and strengthens vulnerability monitoring and mitigation for third-party packages by allowing vulnerability data to be automatically ingested, correlated and acted upon.

5.2 AI-assisted development and "vibe-coding"

The use of AI-assisted development tools, particularly when most coding is delegated to AI with little or no human review (sometimes known as "vibe coding"), introduces new considerations in the secure use of package managers. With this approach, package selection and integration decisions may be partially or fully delegated to large language models. This can reduce the visibility or understanding on why specific packages were selected and how they are integrated. While AI-assisted development is increasingly used in practice, the implications for the potential introduction of insecure, unnecessary, outdated or vulnerable dependencies must be considered. Moreover, this approach also introduces new attack vectors like 'slopsquatting' ([75]), where attackers publish packages matching hallucinated package names generated by AI tools.

As a starting point, to address potential risks introduced by AI-assisted software development, the following measures can be considered:

- maintain clear visibility over all packages that were selected and integrated through AI-assisted or automatically generated code.
- review the necessity and scope of these packages.
- automate selection and integration controls, such as those presented in Sections 4.1 and 4.2, where possible (e.g. within CI/CD pipelines).
- strengthen vulnerability assessment and mitigation processes, since detection and response become more critical when upfront scrutiny is reduced.

([74]) CSAF, 'Common Security Advisory Framework (CSAF)', n.d., https://www.csaf.io/, accessed 1 February 2026.
([75]) Mend.io, 'The Hallucinated Package Attack: Slopsquatting', 15 August 2025, https://www.mend.io/blog/the-hallucinated-package-attack-slopsquatting/, accessed 1 February 2026.

TP-01-26-005-EN-N

ABOUT ENISA

The European Union Agency for Cybersecurity, ENISA, is the Union's agency dedicated to achieving a high common level of cybersecurity across Europe. Established in 2004 and strengthened by the EU Cybersecurity Act, the European Union Agency for Cybersecurity contributes to EU cyber policy, enhances the trustworthiness of ICT products, services and processes with cybersecurity certification schemes, cooperates with Member States and EU bodies, and helps Europe prepare for the cyber challenges of tomorrow. Through knowledge sharing, capacity building and awareness raising, the Agency works together with its key stakeholders to strengthen trust in the connected economy, to boost resilience of the Union's infrastructure, and, ultimately, to keep Europe's society and citizens digitally secure. More information about ENISA and its work can be found here: www.enisa.europa.eu.

ENISA
European Union Agency for Cybersecurity

Athens Office
Agamemnonos 14
Chalandri 15231, Attiki, Greece

Brussels Office
Rue de la Loi 107
1049 Brussels, Belgium

enisa.europa.eu

Publications Office
of the European Union