

TLP - CLEAR



EUROPEAN UNION AGENCY
FOR CYBERSECURITY

ENISA Technical Advisory for Secure Use of Package Managers

DECEMBER 2025

Document History

Date	Version	Modification	Author
Dec 15 2025	0.8	Draft for public consultation	Yonas Leguesse, Razvan Gavrilă

DRAFT

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

For contacting the authors please use product_security@enisa.europa.eu.

For media enquiries about this paper, please use press@enisa.europa.eu.

AUTHORS

Yonas LEGUESSE, ENISA

Razvan GAVRILA, ENISA

ACKNOWLEDGEMENTS

Apostolos MALATRAS, ENISA

Cosmin CIOBANU, ENISA

Ioanna GEORGIADOU, ENISA

Prokopios DROGKARIS, ENISA

LEGAL NOTICE

This publication represents the views and interpretations of ENISA, unless stated otherwise. It does not endorse a regulatory obligation of ENISA or of ENISA bodies pursuant to the Regulation (EU) No 2019/881.

ENISA has the right to alter, update or remove the publication or any of its contents. It is intended for information purposes only and it must be accessible free of charge. All references to it or its use as a whole or partially must contain ENISA as its source.

Third-party sources are quoted as appropriate. ENISA is not responsible or liable for the content of the external sources including external websites referenced in this publication. Neither ENISA nor any person acting on its behalf is responsible for the use that might be made of the information contained in this publication. ENISA maintains its intellectual property rights in relation to this publication.

COPYRIGHT NOTICE

© European Union Agency for Cybersecurity (ENISA), 2025

This publication is licenced under CC-BY 4.0 "Unless otherwise noted, the reuse of this document is authorised under the Creative Commons Attribution 4.0 International (CC BY 4.0) licence (<https://creativecommons.org/licenses/by/4.0/>). This means that reuse is allowed, provided that appropriate credit is given and any changes are indicated".

For any use or reproduction of photos or other material that is not under the ENISA copyright, permission must be sought directly from the copyright holders.

Table of Contents

1. Scope and Context	5
2. Solution Architecture and Data Flows	6
2.1 How Package Managers Work	6
2.1.1 Core Elements	6
2.1.2 Example Flow	7
2.1.3 Package Import and Usage	8
2.2 Package Manager Benefits	10
3. Security Risks in Package Consumption	11
3.1 Packages with Inherent Vulnerabilities	11
3.1.1 Poor Coding or Design Practices and Misconfigurations	11
3.1.1 Discontinued or Unmaintained Packages	12
3.2 Supply Chain Attacks	12
3.2.1 Insertion of Malicious Packages/Dependencies	13
3.2.2 Compromised Legitimate Packages	13
3.2.3 Typosquatting	14
3.2.4 Namespace/Dependency Confusion	14
3.3 Consequences	15
4. Best Practices—Secure Package Consumption	16
4.1 Package selection	17
4.1.1 Selection Recommendations	17
4.1.2 Cheat Sheet	17
4.2 Package integration	19
4.2.1 Integration Recommendations	19
4.2.2 Cheat Sheet	19
4.3 Package Monitoring	20
4.3.1 Package Monitoring Recommendations	20
4.3.2 Cheat Sheet	21
4.4 Vulnerability mitigation	22
4.4.1 Vulnerability Mitigation Steps	22
4.4.2 Cheat Sheet	22
4.5 Automation	23

1. Scope and Context

Modern software development relies heavily on external packages, libraries, and tools obtained through package managers such as the *npm* registry¹, 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, and often open-source code. However, as recent events have demonstrated⁴⁵⁶, they also introduce supply chain risks⁷.

- This document focuses on how developers can securely use package managers as part of their software development lifecycle (SDLC). 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 how to address 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. It does not cover the secure publication of packages or secure coding practices. Examples listed in this document often reference popular ecosystems and tools like Node.JS's *npm* package manager and code repository platforms, such as *GitHub*, but the main principles apply across package ecosystems and code repositories.

¹ <https://www.npmjs.com/>, Accessed November 28, 2025

² <https://pypi.org/project/pip/>, Accessed November 28, 2025

³ <https://maven.apache.org/repositories/index.html>, Accessed November 28, 2025

⁴ <https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/>, Accessed November 28, 2025

⁵ <https://www.aikido.dev/blog/xrp-supplychain-attack-official-npm-package-infected-with-crypto-stealing-backdoor>, Accessed November 28, 2025

⁶ <https://www.wiz.io/blog/shai-hulud-2-0-ongoing-supply-chain-attack>, Accessed November 28, 2025

⁷ <https://www.enisa.europa.eu/sites/default/files/publications/ENISA%20Threat%20Landscape%20for%20Supply%20Chain%20Attacks.pdf>, Accessed November 28, 2025

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.

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

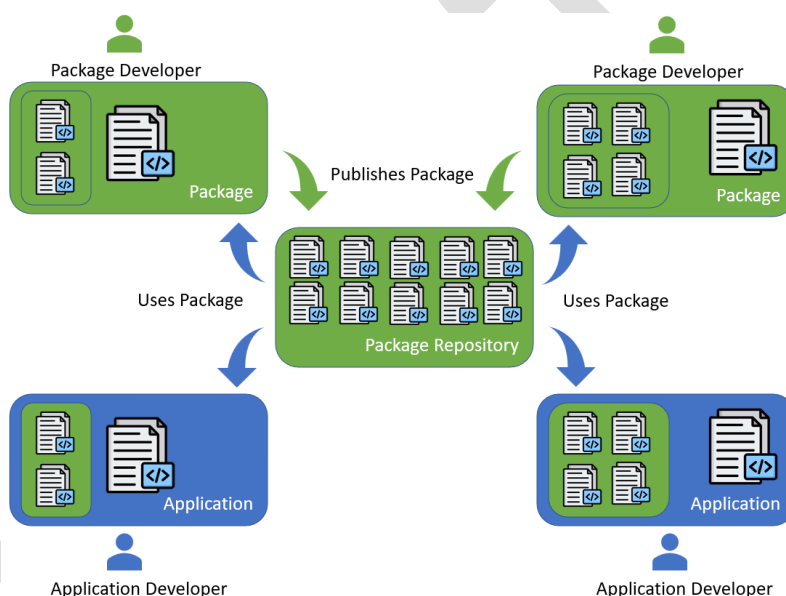
2.1.1 Core Elements

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.

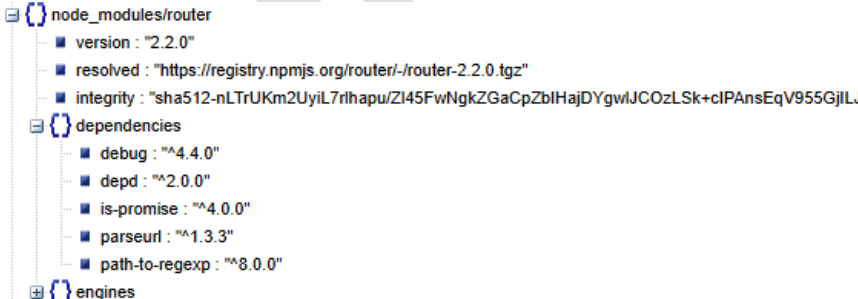
The diagram below illustrates how package developers publish reusable code modules to a shared repository, allowing application developers to download and integrate these packages into their own projects.



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.

Action	Example
Developer writes a simple Node.js app.	<pre> JS index.js X Welcome JS index.js > ... 1 const express = require('express'); 2 const app = express(); 3 4 app.get('/', (req, res) => res.send('Hello from Express!')); 5 6 const port = 3000; 7 app.listen(port, () => { 8 console.log('Listening on http://localhost:\${port}'); 9 }); 10 </pre>

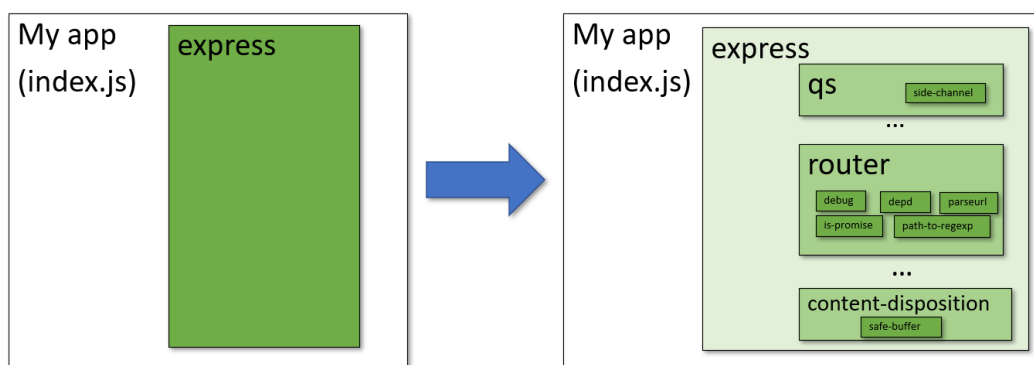
<p>Developer adds the express package to the Node.js project using <code>npm install express</code>.</p> <p>(npm downloads the package from the registry).</p>	<pre>Node.js v18.14.0 PS C:\Users\Documents\vscode_projects\simple_nodejs> npm install express added 68 packages, and audited 69 packages in 4s</pre>
<p>The package express itself depends on other packages (e.g., router, qs, depd).</p> <p>Note: the command <code>npm view <package> dependencies</code> shows the direct dependencies.</p>	<pre>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',</pre>
<p>A lockfile (package-lock.json) records the exact versions of all installed dependencies, including transitive dependencies (packages that your project relies on indirectly).</p>	 <pre>node_modules/router ├─ version: "2.2.0" ├─ resolved: "https://registry.npmjs.org/router/-/router-2.2.0.tgz" ├─ integrity: "sha512-nLTrUKm2UyiL7rhapu/ZI45FwNgkZGaCpZblHajDYgwIJC0zLSk+clPAnsEqV955GjILJn ├─ 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</pre>

This process seems straight forward 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

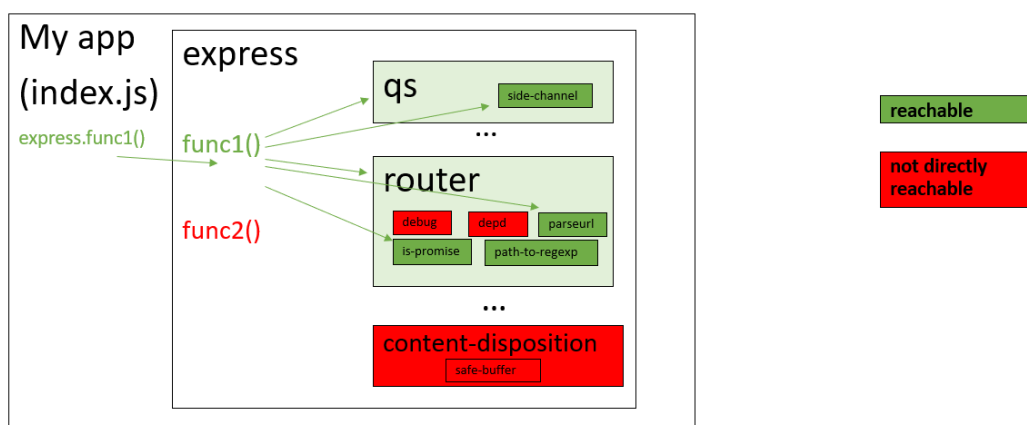
Importing a package, does not just add that single module. It also includes all of its direct and transitive dependencies. In our example, importing *express* brings in around **27** direct dependencies, but a total of **68** packages are installed once all transitive dependencies are resolved.

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.



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 endpoints⁸, 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.



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 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.

⁸ **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

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.

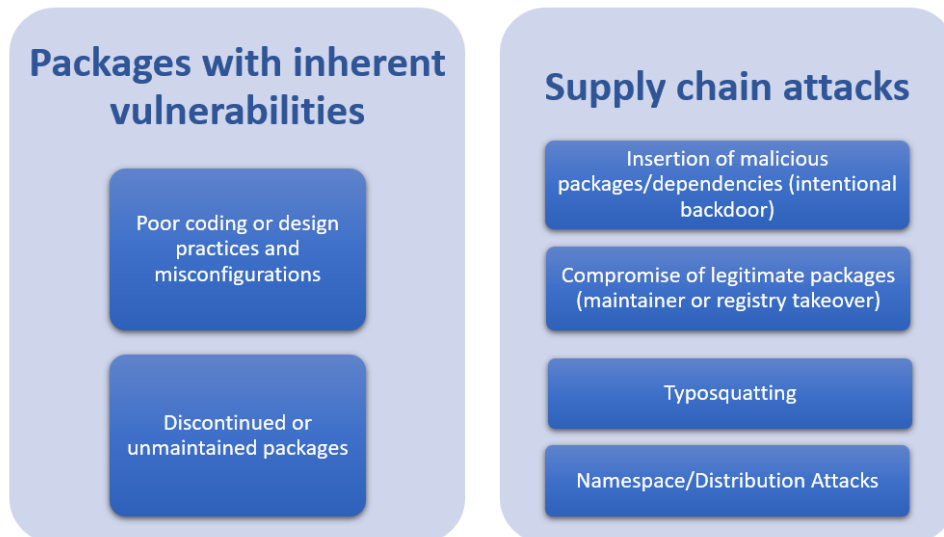
2.2 Package Manager Benefits

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



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)⁹, improper limitation of pathname (CWE-22)¹⁰, sensitive information exposure (CWE-200)¹¹, and deserialization 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` package¹² contains a known critical deserialization of untrusted data (CWE-502) vulnerability¹³ 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.

3.1.1 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* incident¹⁴, which stemmed from the abandoned Rust `tokio-tar` library¹⁵, 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` library¹⁶, 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.

3.2 Supply Chain Attacks

A supply chain refers to the ecosystem of processes, people, organizations, and distributors involved in the creation and delivery of a final solution or product¹⁷. The supply chain can be understood in terms of four key elements:

- **Suppliers:** e.g., package maintainers
- **Supplier assets:** e.g., source code, build pipelines, publishing credentials, and released packages
- **Customers:** e.g., application developers who consume the third-party packages
- **Customer assets:** e.g., the consuming application, its environment, data, and end-users

⁹ <https://nvd.nist.gov/vuln/detail/cve-2021-22931>, Accessed November 28, 2025

¹⁰ <https://nvd.nist.gov/vuln/detail/CVE-2025-27210>, Accessed November 28, 2025

¹¹ <https://nvd.nist.gov/vuln/detail/cve-2023-45143>, Accessed November 28, 2025

¹² <https://www.npmjs.com/package/node-serialize>, Accessed November 28, 2025

¹³ <https://euvd.enisa.europa.eu/vulnerability/CVE-2017-5954>, Accessed November 28, 2025

¹⁴ <https://edera.dev/stories/tarmageddon>, Accessed November 28, 2025

¹⁵ <https://euvd.enisa.europa.eu/vulnerability/CVE-2025-62518>, Accessed November 28, 2025

¹⁶ <https://www.npmjs.com/package/crypto-js>, Accessed November 28, 2025

¹⁷ Beamon, B. M. (1998). Supply chain design and analysis: Models and methods. *International journal of production economics*, 55(3), 281-294.

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 customer¹⁸. 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 credentials¹⁹ cryptocurrency mining, or performing other malicious activities.

An example of an inherently malicious package is *crypto-encrypt-ts*²⁰. 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 development environments. For example, several Visual Studio Code extensions were recently found contain malware (GlassWorm), directly compromising developer machines through their IDE²¹. This approach follows a similar supply-chain distribution approach through 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.

¹⁸ Lella, Ifigeneia, et al., eds. Enisa threat landscape for supply chain attacks. ENISA, 2021.

¹⁹ <https://www.sonatype.com/blog/open-source-malware-index-q2-2025>, Accessed November 28, 2025

²⁰ <https://hackread.com/npm-malware-crypto-wallets-mongodb-turkey-code/>, Accessed November 28, 2025

²¹ <https://thehackernews.com/2025/11/glassworm-malware-discovered-in-three.html>, Accessed November 28, 2025

The *event-stream* incident²² 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 malware²³. 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 preinstall and postinstall scripts, enabling automatic execution of the payload during installation²⁴.

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* package²⁵, which purposely chose a name similar to the popular legitimate *cross-env* package²⁶ show how a single mistyped letter can lead to unintentional inclusion of malicious code through misleading package instances²⁷.

3.2.4 Namespace/Dependency Confusion

Namespace²⁸ or dependency²⁹ confusion is another package distribution attack that exploits differences between internal and public repositories. 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.

²² <https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident>, Accessed November 28, 2025

²³ <https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/>, Accessed November 28, 2025

²⁴ <https://www.sonatype.com/blog/npm-project-used-by-millions-hijacked-in-supply-chain-attack>, Accessed November 28, 2025

²⁵ <https://www.npmjs.com/package/crossenv>, Accessed November 28, 2025

²⁶ <https://www.npmjs.com/package/cross-env>, Accessed November 28, 2025

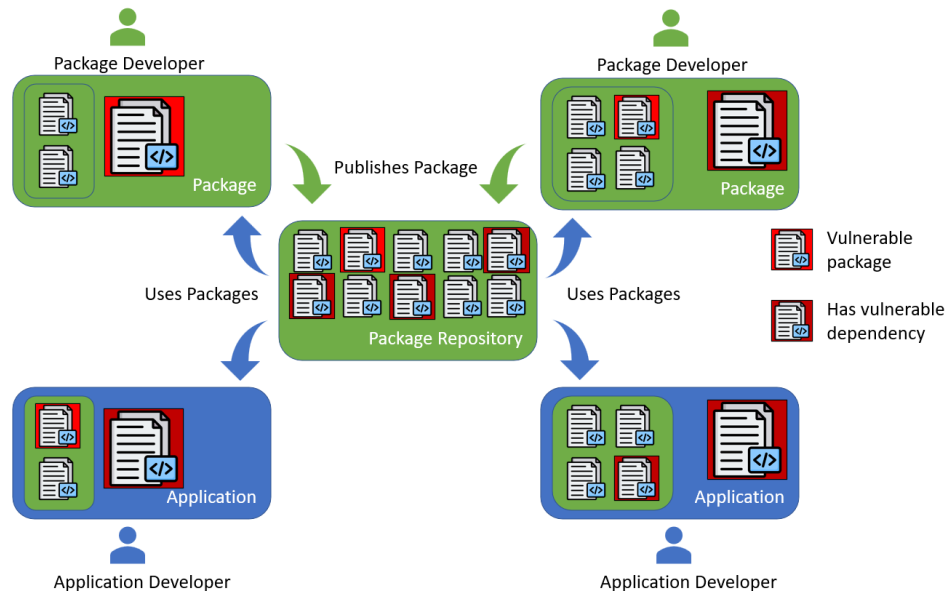
²⁷ <https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry>, Accessed November 28, 2025

²⁸ <https://help.sonatype.com/en/namespace-confusion-protection.html>, Accessed November 28, 2025

²⁹ <https://www.blazeinfosec.com/post/dependency-confusion-exploitation/>, Accessed November 28, 2025

3.3 Consequences

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 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 week³⁰, demonstrated how a single coordinated attack can impact a large portion of the Node.JS ecosystem.

Similarly, the recent critical React vulnerability³¹ (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 affected³², highlighting the scale at which a vulnerability in a widely adopted dependency can propagate.

³⁰ <https://www.paloaltonetworks.com/blog/cloud-security/npm-supply-chain-attack/>, Accessed November 28, 2025

³¹ <https://react.dev/blog/2025/12/03/critical-security-vulnerability-in-react-server-components>, Accessed December 11, 2025

³² <https://www.hackerone.com/blog/cve-2025-55182-react-exploit>, Accessed December 11, 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 lifecycle, 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 lifecycle:

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, isolate, or remove 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 lifecycle. 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.

Finally, since these practices should scale to large and evolving projects, a concluding section discusses automation strategies for integrating these controls into CI/CD pipelines.

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.

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, Synk, 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.	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
Documentation and Transparency	Review documentation and security contact details to confirm openness and accountability.	3.1, 3.2
Typosquat / Name Check	Verify package names carefully to avoid malicious imitations or naming collisions.	3.2.3

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.

Selection Recommendation	Examples
Trusted Source	Several package managers allow developers to publish using provenance statements ³³ to prove that a package was built by a trusted publisher ³⁴ .

³³ <https://docs.npmjs.com/generating-provenance-statements>

³⁴ <https://docs.npmjs.com/trusted-publishers>

	<ul style="list-style-type: none"> Verify the package's provenance to confirm where and how it was published, and that it was released by a trusted and authorised publisher³⁵: <ul style="list-style-type: none"> Look for green check mark on the dedicated npm package web-page <i>npm audit signatures</i> Prefer packages that include valid provenance metadata
Existing Known Vulnerabilities	<ul style="list-style-type: none"> Run scans prior to installation or during dependency review: <ul style="list-style-type: none"> <i>npm audit --json</i> <i>osv-scanner -r .</i> <i>dependency-check --project "My Project" --scan ./</i>³⁶ Consult public databases such as EUVD³⁷, OSV.dev³⁸, Snyk³⁹, or NVD⁴⁰ to check the package's vulnerability history Avoid or postpone adoption of packages with unresolved high/critical severity issues.
Package Signing and Integrity Verification	<ul style="list-style-type: none"> Check that the ecosystem supports integrity metadata, such as SHA-512 hashes in npm (package-lock.json) or enforceable hashes in pip (<i>pip install --require-hashes</i>). 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 <ul style="list-style-type: none"> <i>npm audit signatures</i> reports "missing provenance". Avoid packages that use preinstall or postinstall scripts to download additional code or binaries from external URLs. <ul style="list-style-type: none"> <i>npm view <package> scripts</i> <i>grep -R "curl\ wget\ git clone" node_modules/*/package.json</i> <i>grep -R "curl\ wget\ git clone" node_modules/.</i>
Maintainer Reputation	<ul style="list-style-type: none"> Review maintainer metadata, e.g.: <ul style="list-style-type: none"> <i>npm view <package> contributors</i> <i>npm view <package> maintainers</i> check npm and/or code repository (e.g. github/gitlab) webpage info <i>npm info express -json</i> (maintainers, author, contributors, publisher – most recent publisher) Be cautious of packages owned by newly created, single project accounts, with no additional contributors. <ul style="list-style-type: none"> <i>https://registry.npmjs.org/-/v1/search?text=maintainer:<enter maintainer ID here></i> Prefer packages from verified publishers (Github/npm) and those that include valid provenance metadata
Popularity and Maintenance	<ul style="list-style-type: none"> Check for recent community engagement: <ul style="list-style-type: none"> Check github insights tab (contributors, commits,...) Check CHANGELOG.md and SECURITY.md on repository webpage Check open or recently closed issues and pull requests on repository webpage <i>curl -s "https://api.github.com/repos/<owner>/<repo>/issues?state=all&per_page=100" jq '.[] .state, .created_at, .closed_at'</i> Prefer active projects with frequent commits and tagged releases, e.g.: <ul style="list-style-type: none"> <i>git shortlog -sne</i> <i>npm info <package> -json</i> (time, versions) Prefer dependencies maintained by verified and known active organisations or foundations
Usage of Secure Practices	<ul style="list-style-type: none"> Inspect the package's lifecycle scripts for potentially unsafe commands such as preinstall or postinstall: <ul style="list-style-type: none"> <i>npm view <package> scripts</i> Review dependency trees to confirm the package introduces only necessary dependencies and no excessive or nested components:

³⁵ <https://docs.npmjs.com/viewing-package-provenance>

³⁶ OWASP Dependency-check supports Node.js but is primarily designed for Maven/Java projects.

³⁷ <https://euvd.enisa.europa.eu/>, Accessed November 28, 2025

³⁸ <https://osv.dev/list?ecosystem=npm>, Accessed November 28, 2025

³⁹ <https://security.snyk.io/vuln/npm>, Accessed November 28, 2025

⁴⁰ <https://nvd.nist.gov/>, Accessed November 28, 2025

- 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 lifecycle.

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 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. Note: 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 a Software Bill of Materials during build: <ul style="list-style-type: none"> • <code>syft . -o spdx-json > sbom.json⁴¹</code> • <code>@cyclonedx/cyclonedx-npm --output-file sbom.json⁴²</code>
Vulnerability checks	Block build/install when vulnerabilities are found:

⁴¹ <https://github.com/anchore/syft>, Accessed November 28, 2025

⁴² <https://github.com/CycloneDX/cyclonedx-node-npm>, Accessed November 28, 2025

	<ul style="list-style-type: none"> <code>npm audit --omit=dev --audit-level=high⁴³</code> <code>grype sbom::./sbom.json --fail-on High⁴⁴</code>
Integrity enforcement	<p>npm automatically enforces integrity by verifying SHA-512 hashes stored in package-lockfile.json. Other ecosystems may require explicitly enforcement. For example, with pip::</p> <ul style="list-style-type: none"> <code>pip install --require-hashes -r requirements.txt⁴⁵</code>
Package source enforcement/verification	<p>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.</p> <p>Configure .npmrc file to restrict installations to the trusted registry:</p> <ul style="list-style-type: none"> <code>echo "registry=https://<registry>" >> .npmrc</code>
Installation script prevention	<p>Disable the execution of installation scripts from third-party packages:</p> <ul style="list-style-type: none"> Run installs with scripts disabled: <ul style="list-style-type: none"> <code>npm install --ignore-scripts <package>⁴⁶</code> Set this as a project or global policy: <ul style="list-style-type: none"> <code>echo "ignore-scripts=true" >> .npmrc</code> <code>npm config set ignore-scripts true -g</code> <p>Note: Disabling scripts may impact packages and functionality. This control may be more suited for high-security or isolated environments.</p>
Pinning versions	<ul style="list-style-type: none"> Use a lockfile (package-lock.json⁴⁷, yarn.lock, etc.) to maintain exact dependency versions. Commit both package.json and the lockfile to source control. Enforce lockfile usage in CI/CD⁴⁸: <ul style="list-style-type: none"> <code>npm ci</code> Review changelogs and release notes before upgrading dependencies. Optionally, install packages at a specific date to avoid newly published versions: <ul style="list-style-type: none"> <code>npm install express --before="\$(date -d '-7 days')"</code> <p>Note: Pinning versions can prolong exposure to known vulnerabilities if updates are not regularly reviewed and applied.</p>

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

⁴³ <https://docs.npmjs.com/cli/v8/commands/npm-audit>, Accessed November 28, 2025

⁴⁴ <https://github.com/anchore/grype>, Accessed November 28, 2025

⁴⁵ <https://pip.pypa.io/en/stable/topics/secure-installs/>, Accessed November 28, 2025

⁴⁶ https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html, Accessed November 28, 2025

⁴⁷ <https://docs.npmjs.com/cli/v8/configuring-npm/package-json>, Accessed November 28, 2025

⁴⁸ https://cheatsheetseries.owasp.org/cheatsheets/NPM_Security_Cheat_Sheet.html, Accessed November 28, 2025

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. E.g., <ul style="list-style-type: none"> on new CVEs affecting locked versions on deprecated releases 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 <ul style="list-style-type: none"> <code>grype sbom:./sbom.json⁴⁹</code> <code>osv-scanner scan --sbom.json⁵⁰</code>
Automate vulnerability scanning in CI/CD	Integrate dependency scanning commands into pipeline stages <ul style="list-style-type: none"> <code>npm audit --omit=dev --audit-level=high</code> <code>osv-scanner -r .</code> Schedule periodic scans and fail builds if new high-severity vulnerabilities are detected
Track CVEs/advisories	Subscribe to and monitor public vulnerability feeds: <ul style="list-style-type: none"> EUVD⁵¹, OSV.dev⁵², Snyk⁵³, or NVD⁵⁴ Use GitHub Dependabot⁵⁵ or npm audit⁵⁶ to monitor and notify of new issues affecting dependencies.
Monitor outdated version release	Monitor offset between installed and available versions: <ul style="list-style-type: none"> <code>npm outdated</code>
Set Alerts	Define items to monitor for alerts, e.g.: <ul style="list-style-type: none"> New CVEs affecting locked versions <ul style="list-style-type: none"> GitHub Dependabot alerts⁵⁷ <code>grype sbom:./sbom.json</code> <code>osv-scanner scan --sbom.json</code> Deprecated or unmaintained releases

⁴⁹ <https://github.com/anchore/grype>, Accessed November 28, 2025

⁵⁰ <https://github.com/google/osv-scanner>, Accessed November 28, 2025

⁵¹ <https://euvd.enisa.europa.eu/>, Accessed November 28, 2025

⁵² <https://osv.dev/list?ecosystem=npm>, Accessed November 28, 2025

⁵³ <https://security.snyk.io/vuln/npm>, Accessed November 28, 2025

⁵⁴ <https://nvd.nist.gov/>, Accessed November 28, 2025

⁵⁵ <https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts>, Accessed November 28, 2025

⁵⁶ <https://docs.npmjs.com/cli/v9/commands/npm-audit>, Accessed November 28, 2025

⁵⁷ <https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts>, Accessed November 28, 2025

	<ul style="list-style-type: none"> ○ <code>npm view <package> deprecated</code> ○ <code>npm info <package> time.modified</code> • Maintainer or ownership changes <ul style="list-style-type: none"> ○ <code>npm view <package> maintainers --json</code> <p>Configure alerts using internal notification process (e.g. email, Slack, or Teams) and integrate alerting into CI/CD.</p>
--	---

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 analysis (SCA) 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.

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 related 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	<p>For identified vulnerabilities:</p> <ul style="list-style-type: none"> • Retrieve and record vulnerability metrics (CVSS, EPSS, KEV, VEX):

	<ul style="list-style-type: none"> ○ CVSS: Severity score and vector (e.g., OSV.dev, NVD, Snypk, or EUVD) ○ EPSS⁵⁸: Probability of exploitation score ○ KEV⁵⁹: Check if CVE is listed in CISA's Known Exploited Vulnerabilities catalogue. ○ VEX ⁶⁰data: Vulnerability Exploitability eXchange format used by software suppliers to describe whether vulnerabilities affect a product. • Perform reachability/context analysis (e.g. CodeQL⁶¹, Semgrep⁶², call graph/static analysis tools, or commercial other tools like snyk ⁶³code) to confirm if vulnerable code is actually used.
Prioritise	<ul style="list-style-type: none"> • Define risk thresholds (e.g. "treat CVSS \geq 7.0 or EPSS \geq 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.
Mitigate	<ul style="list-style-type: none"> • Patch or upgrade: <ul style="list-style-type: none"> ○ <code>npm update <package></code> ○ <code>npm install <package> @<version></code> • 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	<ul style="list-style-type: none"> • Categorise vulnerabilities as exploitable, non-exploitable, or mitigated by configuration. <ul style="list-style-type: none"> ○ VEX Product Statuses: <i>known_affected</i>, <i>known_not_affected</i>, <i>under_investigation</i>, and <i>fixed</i>. • Generate/update a VEX mitigation statement. • Update SBOM: <ul style="list-style-type: none"> ○ <code>syft . -o spdx-json > sbom.json</code> ○ <code>@cyclonedx/cyclonedx-npm --output-file sbom.json</code> • Update release notes • Notify relevant internal/external stakeholders (e.g., Impacted customers, relevant authorities, DevSecOps, risk management, compliance teams).

4.5 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
- Gype 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 Gype have public examples illustrating how to integrate SBOM generation and vulnerability scanning into CI/CD workflows.⁶⁴ Similarly, other open-source and commercial solutions often include automation features that help enforce these controls at different stages of the software lifecycle.

⁵⁸ https://www.first.org/epss/data_stats, Accessed November 28, 2025

⁵⁹ <https://www.cisa.gov/known-exploited-vulnerabilities-catalog>, Accessed November 28, 2025

⁶⁰ <https://cyclonedx.org/capabilities/vex>, Accessed November 28, 2025

⁶¹ <https://codeql.github.com/>, Accessed November 28, 2025

⁶² <https://github.com/semgrep/semgrep>, Accessed November 28, 2025

⁶³ <https://docs.snyk.io/manage-risk/prioritize-issues-for-fixing/reachability-analysis>, Accessed November 28, 2025

⁶⁴ <https://anchore.com/blog/javascript-sbom-generation/>, Accessed November 28, 2025

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

