UDC 004.056.53 doi: https://doi.org/10.20998/3083-6298.2026.01.06

Oleksandr Sapalskyi[1], OleksandrZakovorotnyi[1], Valentin Noskov[1]

[1] National Technical University “Kharkiv Polytechnic Institute”, Kharkiv, Ukraine

NPM ARCHITECTURE AS AN ATTACK SURFACE

Abstract. Topicality. The npm ecosystem is a critical infrastructure layer of modern JavaScript/TypeScript development, and its security risks scale rapidly through automated dependency resolution, transitive dependencies, and installation-time script execution. As a result, supply-chain incidents in npm can affect a large number of downstream projects and CI/CD pipelines. **The subject of study** is the npm ecosystem as an attack surface, with a focus on architecture-dependent threat mechanisms related to package publication, package identity and resolution, publisher trust, installation-time execution, and dependency reproducibility and provenance. **The purpose of the** article is to provide an architecture-based analytical review of npm-specific supply-chain threats, to systematize them into a practical taxonomy, and to formulate a layered mitigation approach for engineering teams. **Results obtained**. The article explains the security-relevant properties of npm architecture (registry, CLI, publication, dependency resolution, and installation workflow), proposes an architecture-centered taxonomy of recurring threat mechanisms, analyzes representative incidents as empirical illustrations of these mechanisms, and summarizes protective measures as a layered control model covering dependency intake, deterministic builds, automated monitoring, CI/CD policy enforcement, and publisher hardening. **Conclusion.** Effective risk reduction in npm requires not a single tool, but coordinated technical and organizational controls across the dependency lifecycle; in particular, reproducible installation practices, selective control of lifecycle script execution, continuous dependency monitoring, and stronger publisher authentication and provenance mechanisms improve resilience and auditability without blocking delivery.

Keywords: npm ecosystem security; software supply chain; package confusion; malicious packages; dependency vulnerability scanning; provenance attestation; CI/CD hardening.

Introduction

Problem relevance. Npm is the de facto standard for package management in the JavaScript/TypeScript ecosystem and a key infrastructure component of modern web development. Its official documentation describes npm as the world's largest software registry, used by developers worldwide. npm includes not only the registry but also a CLI and a web interface for managing packages and publishing.

The relevance of this topic is reinforced by the scale of the JavaScript/TypeScript ecosystem itself. According to the Stack Overflow Developer Survey 2025, JavaScript remains the most widely used language among respondents (66%), and TypeScript is also among the most used technologies (43.6%) [1]. This means that infrastructure risks in the npm ecosystem affect a significant proportion of modern web projects and toolchains.

Therefore, npm security should be considered not only as a question of individual defects in library code, but primarily as a problem of the resilience of the ecosystem for distributing, publishing, and installing dependencies. The high degree of centralization of practices around npm means that supply chain incidents (such as publisher compromise, malicious updates, typosquatting, or dependency confusion) have a scalable impact and are of high practical significance for developers and organizations. This has also been recognized as a systemic problem for the platform itself, as evidenced by the tightening of authentication and token management policies by GitHub/npm in 2025.

Literature review. Existing research on software package ecosystems shows that supply-chain attacks are not isolated implementation defects, but a systemic consequence of how modern package managers distribute trust across registries, publishers, dependency resolvers, and automated installation workflows. In particular, academic studies of package-manager ecosystems describe recurring attack families and analyze how ecosystem-scale mechanisms (package naming, dependency resolution, and publication practices) enable attacks to propagate beyond a single compromised package [6]. Other studies examine malicious-package campaigns and automated detection approaches at scale, including methods aimed at identifying suspicious package behavior or package metadata patterns in large repositories [11, 12].

At the same time, industry and standards-oriented sources provide operationally important but fragmented knowledge. npm documentation describes platform-specific threats and mitigation mechanisms, including registry trust, token hygiene, and package integrity features [4, 5], while incident reports and advisories document how real campaigns affect downstream users and CI/CD environments in practice [7, 10]. However, the existing literature and guidance are often split between ecosystem-level empirical analysis, malware or abuse detection methods, and operational recommendations. This creates a gap for practitioners who need a single architecture-based view of npm as an attack surface. The present article addresses this gap by combining an npm-specific architectural analysis with a practical taxonomy of threat mechanisms and a layered mitigation model focused on engineering workflows.

The purpose of the research is to explain the basic principles of the npm ecosystem (registry, package publication, dependency resolution, and installation workflow) in order to identify and classify npm-specific vulnerabilities and to analyze the mechanisms by which they operate.

1. NPM architecture

To properly discuss npm vulnerabilities, we must first define what exactly npm means. In the context of this article, npm is not simply the npm install command,

© Sapalskyi O., Zakovorotnyi O., Noskov V., 2026

but an entire ecosystem for distributing and consuming packages. The official documentation explicitly describes npm as a system of three components: website, CLI, and registry, and describes it as the world's largest registry of software packages. It is this multi-component nature that makes npm a relevant subject for analysis from a security perspective: vulnerabilities arise not only in the code of a specific library, but also in the mechanisms for publishing, identifying, resolving, and installing packages [2].

In a typical project, npm begins with package.json, which defines dependencies and rules for selecting them by version. The npm CLI then accesses the registry to resolve packages by name and version, retrieves metadata, and downloads the corresponding artifacts. The npm documentation specifically notes that the CLI interacts with the registry when resolving packages, and that the registry supports not only read operations but also a write API for publishing packages and managing user information [3]. This is fundamentally important for the topic of this article: the same infrastructure is responsible for both package consumption and publishing, meaning that publisher trust issues and package installation issues are architecturally linked.

The key feature of npm as an ecosystem is the automatic construction of a dependency tree, including not only direct but also transitive dependencies. From a practical standpoint, this is the source of the ecosystem's scalability: a developer includes a few packages, but in fact receives a large set of someone else's code, selected and built automatically. In a normal engineering process, this risk is partially controlled through package-lock.json, which, according to npm documentation, captures the exact dependency tree and allows for reproducible installations regardless of intermediate dependency updates. npm also explicitly recommends committing this file to the repository. This is important for security not in itself, but because installation reproducibility reduces the likelihood of unnoticeable dependency drift between production machines, CI, and the production environment.

However, installing a package in npm isn't just a matter of downloading files to "node_modules." folder. The npm model inherently includes a mechanism for scripts and lifecycle events, described in package.json. The npm documentation emphasizes that the "scripts" property supports both arbitrary commands and built-in lifecycle scripts executed within various operations. This means that at the install/ci stage, the npm architecture allows for command execution as a standard part of the process. From a research perspective, this is where an important boundary is drawn: we're not analyzing "bugs in package code," but rather the fact that the installation infrastructure itself allows for code execution within a trusted user or CI environment.

The configuration of the package source plays an additional role. npm uses a public registry by default, but allows for the configuration of compatible alternative registries and custom registries. The documentation also states that authentication parameters and some behavior are tied to a specific registry, and the choice of registry can depend on configuration and context. Consequently, the question of "where exactly the package came from" becomes not just an operational detail, but part of the attack surface: npm's architecture allows for variability in sources and trust settings, which directly relates to risks of dependency confusion and configuration errors.

2. Taxonomy of npm ecosystem vulnerabilities

An analysis of the npm architecture reveals that vulnerabilities arising in this ecosystem are not a random collection of isolated incidents, but rather the result of specific design decisions underlying the package distribution and consumption model. npm integrates the processes of publishing, identifying, resolving dependencies, and installing packages into a single infrastructure, ensuring high usability and scalability of the ecosystem while simultaneously creating a complex system of trust between developers, publishers, and build tools.

Based on the above, this article proposes classifying npm vulnerabilities based on the ecosystem architecture elements through which attacks are implemented. This approach allows us to identify stable groups of vulnerabilities that reflect key trust and automation points in npm, thereby creating a basis for subsequent analysis of real-world incidents and protective measures.

The first group is package identity vulnerabilities. Here, the problem arises at the package name and namespace level: a user or build system requests one identifier and receives another, either superficially similar or conflicting with the private naming convention. This includes typosquatting and dependency confusion. npm explicitly addresses this class of attacks, noting the ability to detect typosquatting while emphasizing that dependency confusion cannot be automatically detected; the use of scoped packages is recommended as a key mitigation measure. From a methodological perspective, this category is important because the attack occurs even before the package contents are analyzed: compromise occurs at the level of deciding what exactly is considered a "relevant package" [4].

The second group is publisher trust vulnerabilities. Here, the attack target is not the package name, but the publishing permissions. npm specifically identifies account takeover as one of the primary vectors in its Threats and Mitigations section, describing both classic password compromise scenarios and more specific cases, such as risks associated with expired email domains. This is crucial for the npm ecosystem: if an attacker gains control of the maintainer account, they can publish formally "legitimate" versions of an already trusted package. Therefore, the problem here is not bypassing the code review mechanism, but rather spoofing the trust principal within the standard publishing process.

The third group is installation-time execution risks. In npm, package installation architecturally allows for lifecycle script execution, as the scripting mechanism is part of the normal CLI operating model. The npm documentation describes lifecycle events in detail and specifically notes that when installing a git dependency containing prepare, npm may install its dependencies and devDependencies and then execute prepare before

packaging and installation. This means that the npm install stage is not a passive file delivery; it may involve executing commands in the developer environment or CI. Therefore, this class should be distinguished separately within the classification of npm vulnerabilities: the risk arises from the very fact of permissible code execution during the installation stage, and not from logical defects in the final library.

The fourth group is dependency resolution and package composition reproducibility vulnerabilities. The npm architecture assumes automatic construction of a tree of direct and transitive dependencies, meaning the actual composition of the resulting code is determined not only by explicit developer choices but also by the versioning policy, the registry state, and the tree's lockfile status. In this category, the key risk mechanism is the discrepancy between the expected and actual dependency composition: version drift, uncommitted transitive updates, uncontrolled tree changes between environments. While these phenomena do not always constitute an attack in themselves, they create conditions under which supply-chain attacks become less visible and more difficult to reproduce during investigation. The connection of this category to the npm architecture stems from the basic model of automatic dependency resolution described in the registry and CLI documentation.

The fifth group is integrity/provenance vulnerabilities. This concerns whether the consumer can verify that the received package was not tampered with during delivery, and where exactly it was built and published. npm describes ECDSA registry signatures as a mechanism for detecting package content tampering (for example, when a mirror or proxy is compromised), and is also developing provenance attestations as a way to establish a verifiable link between the package, the source code, and the build/publishing process [5]. At the same time, the documentation explicitly states the limitation of provenance: the presence of provenance does not prove the absence of malicious code, but only makes the package's origin verifiable. Therefore, in the classification, this is not "out-of-the-box protection," but a separate class of risks and controls associated with insufficient chain-of-origin verification.

Table 1 – **Taxonomy of npm vulnerabilities with examples**

Category	What it is	Representative examples
Typosquatting	A malicious package is installed because its name is a typo or visually similar to a legitimate package name.	loadsh instead of lodash
Package substitution / Registry confusion	A resolver installs a public package instead of the intended private/internal package due to name collision and registry resolution behavior.	Internal package acme-utils is referenced unscoped, but CI installs a public acme-utils; private/public name collision with higher public version winning resolution.
Publisher account / token compromise	An attacker gains legitimate publish rights by compromising maintainer credentials, recovery channel, or npm token.	Stolen npm access token used to publish a new version; maintainer account takeover via password reset on an email at an expired domain.
Publish pipeline compromise (CI/CD)	An attacker abuses the automated release workflow and publishes through CI without directly using the maintainer's interactive login.	Leaked CI secret in GitHub Actions; overprivileged publish token in CI; compromised release workflow pushes malicious package version.
Malicious install-time scripts	A package executes attacker-controlled code during installation through npm lifecycle scripts.	Malicious preinstall steals environment variables; postinstall downloads and runs an external payload; prepare runs when installing a git dependency.
Dependency resolution & reproducibility risks	The actual installed dependency tree differs from what maintainers expect, making supply-chain attacks harder to notice and investigate.	Installing from package.json without a stable lockfile yields different transitive versions across machines; version drift after permissive semver ranges (^, ~).
Transitive dependency exposure	Risk enters through indirect dependencies that the team did not explicitly choose or review.	A deeply nested utility package becomes malicious/compromised; an indirect dependency introduces install-time behavior not visible in top-level package.json.
Build / provenance / integrity gaps	Consumers cannot reliably verify artifact origin or detect tampering in the delivery/build chain.	Package is installed without checking registry signatures; no provenance attestation for a published version; artifact origin cannot be tied to a specific trusted build workflow.

3. Brief History of Major npm Supply-Chain Events

The relevance of npm ecosystem security is best understood through incidents that became visible beyond individual projects and forced changes in security practices across the JavaScript/TypeScript community.

The cases below are short historical overview of major events that illustrate how npm supply-chain risk evolved from isolated package compromises into broader, faster, and more automated attack campaigns. npm itself frames the ecosystem threat landscape in terms of account takeovers, malicious package uploads, typosquatting / dependency confusion, and malicious changes to existing packages, which helps explain why these incidents recur in different forms over time.

A foundational case is the event-stream incident (2018). npm's incident report states that its security team was notified of malicious code introduced into the event-stream dependency chain and responded by removing flatmap-stream and event-stream@3.3.6 from the registry, while also taking ownership of the event-stream package to prevent further abuse. npm further described the incident as beginning with a social-engineering attack that resulted in malicious control over package maintenance. Event-stream became a canonical reference point because it showed that a trusted, widely used package (190 million downloads, 3,905 dependent packages) could be weaponized through the normal dependency path, without requiring users to install an obviously suspicious package name [6].

The widespread npm supply-chain compromise alert (September 2025) marked a different stage in the history of npm security: public-sector warning and ecosystem-scale response. CISA's alert described a widespread compromise affecting the npm ecosystem and noted that attackers leveraged automation to spread rapidly by authenticating to the npm registry as compromised developers and injecting code into additional packages [7]. This matters historically because it reflects a transition from high-profile but comparatively contained incidents to campaigns requiring coordinated defensive guidance, including recommendations around version pinning, credential rotation, and stricter release hygiene. In other words, by 2025 the problem had become large enough that response guidance was not only package-specific but ecosystem-operational.

Closely related, but analytically distinct, is the Shai-Hulud self-replicating npm malware campaign (2025). CERT/CC's vulnerability note describes Shai-Hulud as a self-propagating malware variant spreading through credential theft and automated package publishing, with more than 500 affected packages at the time of publication and continued growth thereafter. CERT/CC also explicitly identifies exploitation of known vectors such as credential theft, package impersonation, and automated propagation, and highlights the speed of escalation [8]. The historical significance of Shai-Hulud is that it demonstrates a worm-like pattern in npm supply-chain abuse: compromised developer environments and CI/CD credentials were not only targets, but also mechanisms for further propagation. This made the campaign qualitatively different from earlier one-off package compromises, because the attacker's reach could increase through the ecosystem's own automation channels.

Another major 2025 incident involved malicious Nx package versions. The official Nx postmortem states that on August 26, 2025, malicious versions of several Nx packages were published to npm after attackers exploited a GitHub Actions injection vulnerability to steal an npm publishing token; the malicious releases were available for approximately four hours. The Nx team also reported post-incident changes including migration to npm Trusted Publishers and additional release controls [9]. GitHub's advisory for the same incident documents that malicious versions of nx and related plugins were published and contained code that scanned the filesystem, collected credentials, and posted them to GitHub repositories under user accounts. This case is especially important in a short historical section because Nx is developer tooling, not a niche library; it demonstrates how compromise of build-tooling packages can immediately affect development workstations, CI environments, and organizational secrets.

The eslint-config-prettier compromise (July 2025) is another important example because it affected a widely used developer-facing package and was formally tracked in public advisory systems. GitHub's advisory reports that specific versions of eslint-config-prettier contained embedded malicious code as part of a supply-chain compromise and that installing affected versions executed an install.js file launching malware on Windows. The NVD entry for CVE-2025-54313 records the same affected versions and confirms the install-time execution behavior [10]. This incident is historically relevant not only because of package popularity, but because it reinforced a repeated pattern seen across npm attacks: compromise of maintainership or publishing credentials followed by malicious code delivered through standard installation behavior.

Taken together, these incidents show a clear progression in npm supply-chain risk. Earlier incidents such as event-stream established the core problem of trust abuse in dependency ecosystems; later incidents in 2025 demonstrated increased operational tempo, stronger use of CI/CD and token theft, and large-scale propagation supported by automation. Public alerts from CISA and CERT/CC, together with postmortems from affected maintainers such as the Nx team, indicate that npm supply-chain security is no longer a matter of isolated package anomalies but an ecosystem-level operational concern that affects maintainers, toolchain vendors, and downstream organizations alike.

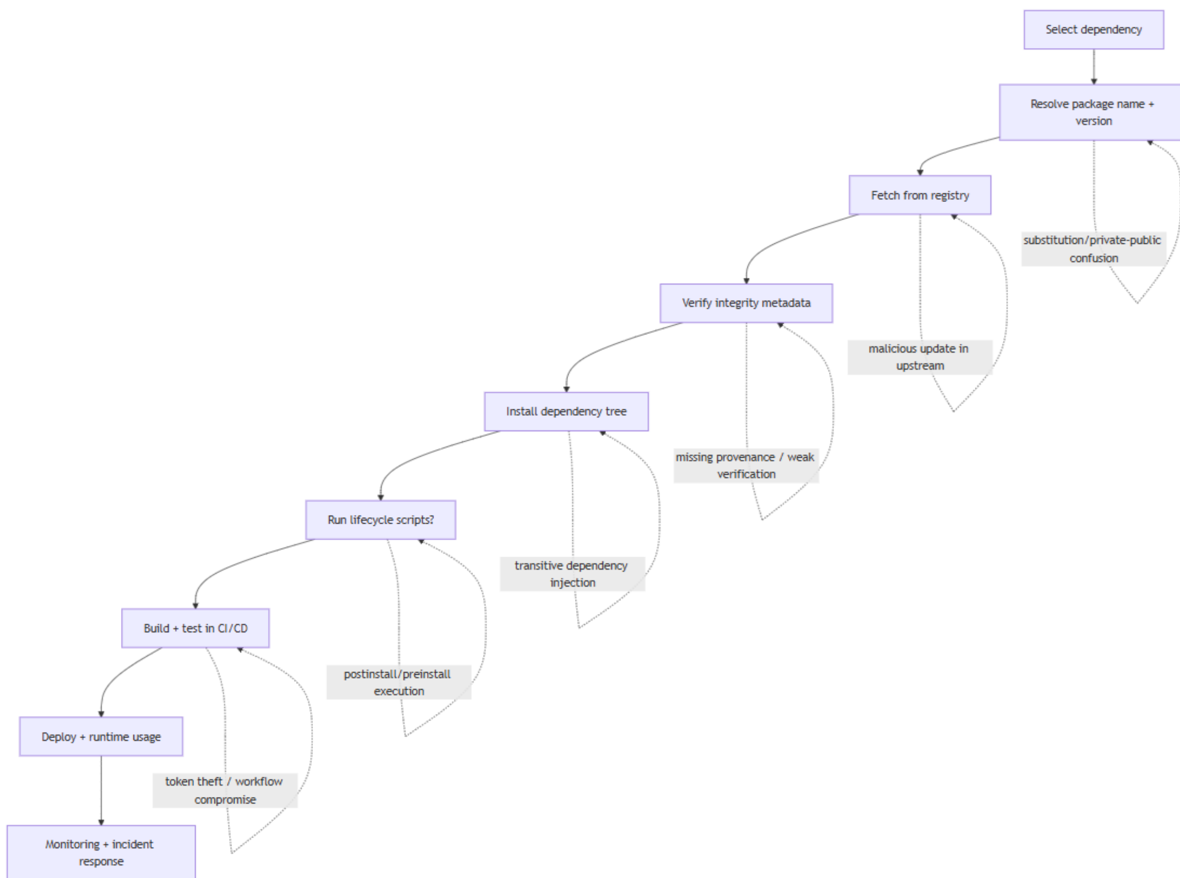

Fig. 1. Risks graph

4. Protective Measures

Effective protection against npm-specific threats should be organized as a layered control system rather than as a single tool or a single check. In practice, the attack surface appears at several stages of the dependency lifecycle: package discovery, package retrieval, dependency resolution, installation, CI/CD execution, and package publishing. Therefore, a robust defense model combines registry-level controls, deterministic build practices, automated vulnerability management, CI policy enforcement, and hardening of maintainer and publishing workflows.

The first layer should be package ingress control. For organizations, it is preferable to avoid direct, unrestricted access from developer machines and CI jobs to the public registry. A repository manager such as Sonatype Nexus Repository can act as an intermediary and supports npm hosted, proxy, and group repositories, including proxying/caching of upstream registry content. This architecture reduces operational dependence on live external fetches and creates a natural enforcement point for internal policy. Sonatype's Repository Firewall further adds an explicit security control at ingress by automatically quarantining suspicious components, and Policy Compliant Component Selection can remove quarantined versions from npm metadata so clients do not resolve them by default.

The second layer is deterministic dependency resolution and installation discipline. npm's package-lock.json records the exact dependency tree and is intended to be committed to source repositories, which makes installations reproducible across environments. In CI/CD, npm ci is the preferred installation mode because it is designed for automated environments, performs a clean install, and fails if the lockfile and package.json are inconsistent instead of silently rewriting the lockfile. This directly reduces the window for accidental drift and makes anomalous dependency changes easier to detect during code review.

A related point is installation-time code execution. npm supports lifecycle scripts, and the npm configuration includes ignore-scripts, which disables execution of scripts declared in package.json during installation (except when a script is explicitly invoked). This should not be treated as a universal default for all builds, because many projects legitimately depend on build/install scripts; however, it is highly useful in selected contexts (for example, metadata inspection pipelines, triage jobs, or high-sensitivity validation stages) where dependency code does not need to execute. In other words, the control should be applied selectively and intentionally, not dogmatically.

The third layer is continuous vulnerability detection and controlled remediation. At minimum, projects should use npm audit (for registry-advisory-based checks) and repository-level alerting in the hosting platform. On GitHub, Dependabot is not one mechanism but a set of coordinated mechanisms: alerts (detection), security updates (automated fix PRs for known vulnerabilities), and version updates (routine dependency currency). GitHub's documentation also states that these features rely on the dependency graph, which is built from manifest and lock files and updated as the repository

changes. This matters operationally: if lockfiles are absent or not maintained, the quality of downstream automation degrades.

Dependabot should be configured, not merely enabled. GitHub documents that version updates are managed through dependabot.yml, and its options support grouping updates into fewer, targeted pull requests. This is important in research and in practice because alert fatigue is a real failure mode: a noisy security program is often an ineffective security program. Grouping, scheduling, and scoped policies make automated updates reviewable, which increases acceptance rates without removing human oversight.

This recommendation is also supported by empirical research on npm dependency update behavior, which shows that update strategies vary across projects and are associated with package characteristics; therefore, dependency updates should be managed as a policy and governance task rather than treated as a purely mechanical automation step [12].

Renovate is a strong complementary tool in teams that need more granular update policy control. Official Renovate documentation provides explicit controls for packageRules, scheduling, lock file maintenance, and vulnerability alert PR behavior [13]. Notably, Renovate supports dedicated vulnerabilityAlerts configuration and documents that such PRs can "skip the line" relative to normal scheduling limits, while lock-file maintenance can be enabled separately and automated. This makes Renovate particularly suitable for large monorepos and organizations that need differentiated policies for production dependencies, developer tooling, internal packages, and high-churn ecosystems.

The fourth layer is merge-time policy enforcement in CI. Detection alone is insufficient if pull requests introducing risky dependencies can still be merged without constraints. GitHub's dependency review action is designed specifically for pull-request dependency diffs and can add enforcement mechanisms to workflows. GitHub also documents common hardening customizations: fail builds based on vulnerability severity, deny specific licenses, and fail on selected dependency scopes. When the workflow is marked as required, dependency changes become subject to the same merge governance as tests and static analysis. This is a materially stronger control than post-merge alerting alone.

The fifth layer is hardening of maintainers and publishing workflows, because many large npm incidents have involved account compromise or misuse of publishing credentials. npm provides organizational 2FA enforcement, and its documentation explicitly notes that this makes it harder for malicious actors to access an organization's packages and settings; it also describes the operational consequence that members without 2FA are removed until they comply. For package publication pipelines, npm's trusted publishing should be preferred over long-lived write tokens: npm documents that trusted publishing uses OIDC, creates a trust relationship with the CI provider, and replaces long-lived reusable credentials with short-lived workflow-specific credentials. npm further recommends using a read-only token for installing private dependencies in CI, while trusted publishing handles the publish step.

Where tokens are still required, least privilege should be enforced. npm's access-token documentation distinguishes granular access tokens with different permission levels and package/account scope controls, which is essential for separating read-only installation access from publication permissions and for reducing blast radius in the event of exposure. In practice, this means avoiding shared all-powerful tokens, limiting write scope to the smallest possible set of packages, and rotating credentials on any suspicion of leakage.

The sixth layer is integrity and provenance verification. npm provides ECDSA registry signatures and documentation for verifying them, including the npm audit signatures workflow, while trusted publishing automatically generates provenance attestations for published packages. These controls do not replace review and policy enforcement, but they significantly improve the ability to verify artifact authenticity and publication origin in automated pipelines. In a research framing, this is a shift from purely vulnerability-centric defense toward supply-chain authenticity guarantees.

Finally, a mature program should support inventorying and incident response. npm documents npm sbom, which generates a software bill of materials in SPDX or CycloneDX format. SBOM generation does not prevent an attack, but it materially improves containment and scoping after disclosure or compromise because affected components can be identified and prioritized more quickly across repositories and releases [14].

Discussion of results

The results of this study support the thesis that npm-related risk should be analyzed primarily as an ecosystem and supply-chain problem, not as a set of isolated failures in individual repositories. The architectural review showed that npm is not only a CLI tool but a combined system that includes the registry, the CLI, and the web platform, operating at very large scale; this makes trust transfer, package publication, dependency resolution, and installation behavior central security concerns at the ecosystem level. npm itself describes the platform as the world's largest software registry and explicitly documents threat classes such as account takeover and malicious package publication (including typosquatting / dependency confusion and malicious modification of existing packages), which reinforces the systemic framing adopted in this article.

A key result of the article is that the architecture-first approach is analytically productive. Once npm is examined as an infrastructure for package distribution and automated dependency consumption, the main threat groups become visible as recurring mechanisms rather than случайные incidents. In this sense, the classification developed in the article is not an arbitrary taxonomy but a structured representation of where trust is created, delegated, and executed in the npm lifecycle. The historical cases then serve as empirical confirmation of these mechanisms: the event-stream incident illustrates compromise through a trusted dependency path, while later campaigns such as Shai-Hulud demonstrate how

credential theft and automated republication can scale the same ecosystem-level weaknesses across hundreds of packages.

Another important outcome is the clarification that the problem is systemic even when an incident begins with a single package or a single maintainer account. The article's incident analysis shows that the impact expands through npm's normal operational properties: transitive dependency use, automation in CI/CD, and rapid package distribution. Therefore, the relevant unit of analysis is not the compromised repository alone, but the dependency graph and the delivery process that connect many downstream projects to the same upstream artifact. This is precisely why public incident descriptions often emphasize affected package counts, published malicious versions, and propagation patterns rather than only the initial compromise point.

The protection section of the article also leads to a clear result: no single control is sufficient. The analysis supports a layered model in which different controls are mapped to different stages of the lifecycle. Repository-gateway controls (e.g., Nexus/Repository Firewall) address ingress and quarantine; lockfiles and npm ci address reproducibility and drift; automated update tooling (Dependabot / Renovate) addresses maintenance and remediation cadence; dependency-review gates address merge-time governance; and trusted publishing plus provenance/signature verification address publisher authentication and artifact authenticity. This layered interpretation is consistent with vendor and platform documentation, which describes these capabilities as complementary rather than interchangeable.

At the same time, the study indicates an important practical constraint: defensive maturity depends not only on technical tooling, but on governance quality. Tools such as Dependabot and dependency review actions provide strong automation, but their effectiveness is determined by configuration discipline, review practice, and enforcement in CI rulesets. Similarly, provenance and signature verification are meaningful only when organizations actually integrate them into build and release workflows. Thus, the article's main practical result is not a list of tools, but a governance model for dependency intake, update approval, and release trust.

Conclusions

Npm has become a de facto standard infrastructure layer for modern JavaScript/TypeScript development. This conclusion is supported by the scale of the ecosystem itself: npm describes its registry as the world's largest software registry, notes that it is used globally (including by organizations for private development), and the npm website states that the registry contains more than two million packages. At the same time, JavaScript and TypeScript remain among the most widely used technologies, which further increases the practical significance of npm-related risks for contemporary software development.

Against this background, the main conclusion of this article is that npm security should be understood as a systemic architectural issue of the ecosystem, rather than as a local problem of a single repository. By examining how npm operates (registry, CLI, publication, dependency resolution, and installation), the study showed that projects are exposed to a recurring set of structured threats generated by the normal trust and automation mechanisms of the ecosystem. The analysis therefore justifies treating npm-related threats as a distinct class of supply-chain risks that require dedicated analytical and operational approaches. The fact that npm itself documents recurring ecosystem attack patterns (e.g., account takeover, malicious package publication, typosquatting / dependency confusion, malicious changes to existing packages) supports this interpretation [4].

An architecture-first perspective makes it possible to move from fragmented incident descriptions to a coherent understanding of threat categories. In this article, the review of npm architecture made it possible to identify and systematize the main categories of threats, while the historical incident section demonstrated that these categories are not abstract constructs but recurring patterns observed in real attacks. Thus, the article's contribution is not limited to listing incidents; it provides a structured framework for interpreting why such incidents continue to reappear in different forms.

Because npm is often used as routine infrastructure and integrated deeply into development workflows, its risk profile may be underestimated in day-to-day engineering practice. The results of this study indicate that working with npm requires explicit responsibility and a deliberate security process. Trust in package ecosystems cannot remain implicit: it must be supported by concrete protective steps across the dependency lifecycle, including controlled package ingress, deterministic installation, dependency governance, CI/CD enforcement, and stronger publishing/authenticity controls. In this sense, the central practical implication of the article is not to reject npm, but to use it under a mature security model appropriate to its scale and criticality.

In summary, npm's ubiquity is precisely what makes its security problem strategically important. The broader the adoption of npm, the greater the downstream effect of architectural weaknesses in package distribution and trust transfer. Therefore, npm-related threats should be treated as an ecosystem-level engineering and governance challenge, and effective mitigation should be built as a systematic set of controls rather than as an ad hoc reaction to individual incidents.

References

1. Stackoverflow. Survey (2025). “Technologies section”, https://survey.stackoverflow.co/2025/technology
2. NPM Documentation. About npm, https://docs.npmjs.com/about-npm
3. NPM Documentation. Registry, https://docs.npmjs.com/cli/v11/using-npm/registry
4. NPM Documentation. Threats and Mitigations, https://docs.npmjs.com/threats-and-mitigations
5. NPM Documentation. About ECDSA registry signatures, https://docs.npmjs.com/about-registry-signatures

6. Duan, R., Alrawi, O., Kasturi, R. P., Elder, R., Saltaformaggio, B. and Lee, W. (2021), “Towards measuring supply chain attacks on package managers for interpreted languages”, *Network and Distributed Systems Security (NDSS) Symposium 2021*, https://dx.doi.org/10.14722/ndss.2021.23055
7. Cybersecurity and Infrastructure Security Agency (CISA). (2025), “Widespread Supply Chain Compromise Impacting NPM Ecosystem”, https://www.cisa.gov/news-events/alerts/2025/09/23/widespread-supply-chain-compromise-impacting-npm-ecosystem
8. CERT Coordination Center (CERT/CC). (2025), “NPM supply chain compromise exposes challenges to securing the ecosystem from credential theft and self-propagation”, *Vulnerability Note VU#534320*, https://kb.cert.org/vuls/id/534320
9. Strumpflohner, J. (2025), “S1ngularity - What Happened, How We Responded, What We Learned”, *Nx Blog*, https://nx.dev/blog/s1ngularity-postmortem
10. National Institute of Standards and Technology (NIST). (2025), “CVE-2025-54313”, *National Vulnerability Database*, https://nvd.nist.gov/vuln/detail/CVE-2025-54313
11. Neupane, S., Holmes, G., Wyss, E., Davidson, D. and De Carli, L. (2023), “Beyond typosquatting: an in-depth look at package confusion”, *In 32nd USENIX security symposium (USENIX security 23)*, https://ldklab.github.io/assets/papers/usenix23-confusion.pdf
12. Javan Jafari, A., Costa, D. E., Shihab, E. and Abdalkareem, R. (2023), “Dependency update strategies and package characteristics”, *ACM Transactions on Software Engineering and Methodology*, Vol. 32(6), pp. 1-29, https://doi.org/10.1145/3603110
13. Renovate Documentation, https://docs.renovatebot.com/getting-started/use-cases
14. Wallen, C. M., Alberts, C. J., Bandor, M. S. and Woody, C. (2023), “Software Bill of Materials Framework: Leveraging SBOMS for Risk Reduction”, https://www.sei.cmu.edu/library/software-bill-of-materials-framework-leveraging-sboms-for-risk-reduction/

Received (Надійшла) 20.02.2026
Accepted for publication (Прийнята до друку) 19.03.2026

ВІДОМОСТІ ПРО АВТОРІВ/ ABOUT THE AUTHORS

Сапальський Олександр Андрійович – аспірант кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Sapalskyi Oleksandr – PhD student, Department of Computer Engineering and Programming, National Technical University ‘Kharkiv Polytechnic Institute’, Kharkiv, Ukraine;
e-mail: Oleksandr.Sapalskyi@cs.khpi.edu.ua; ORCID Author ID: https://orcid.org/0009-0002-5749-8527.

Заковоротний Олександр Юрійович – доктор технічних наук, професор, завідувач кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Oleksandr Zakovorotnyi – Doctor of Technical Sciences, Professor, Head of the Department of Computer Engineering and Programming, National Technical University ‘Kharkiv Polytechnic Institute’, Kharkiv, Ukraine;
e-mail: Oleksandr.Zakovorotnyi@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0003-4415-838X;
Scopus Author ID: https://www.scopus.com/authid/detail.uri?authorId=57201613700.

Носков Валентин Іванович – доктор технічних наук, доцент, професор кафедри комп'ютерної інженерії та програмування, Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Valentyn Noskov – Doctor of Technical Sciences, Associate Professor, Professor of the Department of Computer Engineering and Programming, National Technical University ‘Kharkiv Polytechnic Institute’, Kharkiv, Ukraine;
e-mail: valentyn.noskov@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0002-7879-0706;
Scopus Author ID: https://www.scopus.com/authid/detail.uri?authorId=57331254200.

АРХІТЕКТУРА NPM ЯК ПОВЕРХНЯ АТАКИ

О.А. Сапальський, О.Ю. Заковоротний, В.І. Носков

Анотація. **Актуальність.** Екосистема npm є критичним рівнем інфраструктури сучасної розробки на JavaScript/TypeScript, а її ризики безпеки швидко масштабуються завдяки автоматизованому розв'язанню залежностей, транзитивним залежностям та виконанню скриптів під час встановлення. Як результат, інциденти ланцюга поставок у npm можуть впливати на велику кількість проектів нижче за течією та конвеєрів CI/CD. **Предметом дослідження** є екосистема npm як поверхня атаки, з акцентом на архітектурно-залежних механізмах загроз, пов'язаних з публікацією пакетів, ідентифікацією та роздільною здатністю пакетів, довірою видавця, виконанням під час встановлення, а також відтворюваністю та походженням залежностей. **Мета статті** полягає в тому, щоб надати аналітичний огляд загроз ланцюгу поставок, специфічних для npm, на основі архітектури, систематизувати їх у практичну таксономію та сформулювати багаторівневий підхід до пом'якшення наслідків для інженерних команд. **Отримані результати.** У статті пояснюються властивості архітектури npm, що стосуються безпеки (реєстр, CLI, публікація, вирішення залежностей та робочий процес встановлення), пропонується архітектурно-орієнтована таксономія механізмів повторюваних загроз, аналізуються репрезентативні інциденти як емпіричні ілюстрації цих механізмів та узагальнюються захисні заходи як багаторівнева модель керування, що охоплює прийом залежностей, детерміновані збірки, автоматизований моніторинг, забезпечення дотримання політик CI/CD та посилення захисту видавців. **Висновок.** Ефективне зниження ризиків у npm вимагає не одного інструменту, а скоординованих технічних та організаційних заходів контролю протягом усього життєвого циклу залежностей; зокрема, відтворювані методи встановлення, вибірковий контроль виконання скриптів життєвого циклу, безперервний моніторинг залежностей та сильніші механізми аутентифікації та походження видавця покращують стійкість та аудит без блокування доставки.

Ключові слова: безпека екосистеми npm; ланцюг постачання програмного забезпечення; плутанина з пакетами; шкідливі пакети; сканування вразливостей залежностей; атестація походження; посилення безпеки CI/CD.