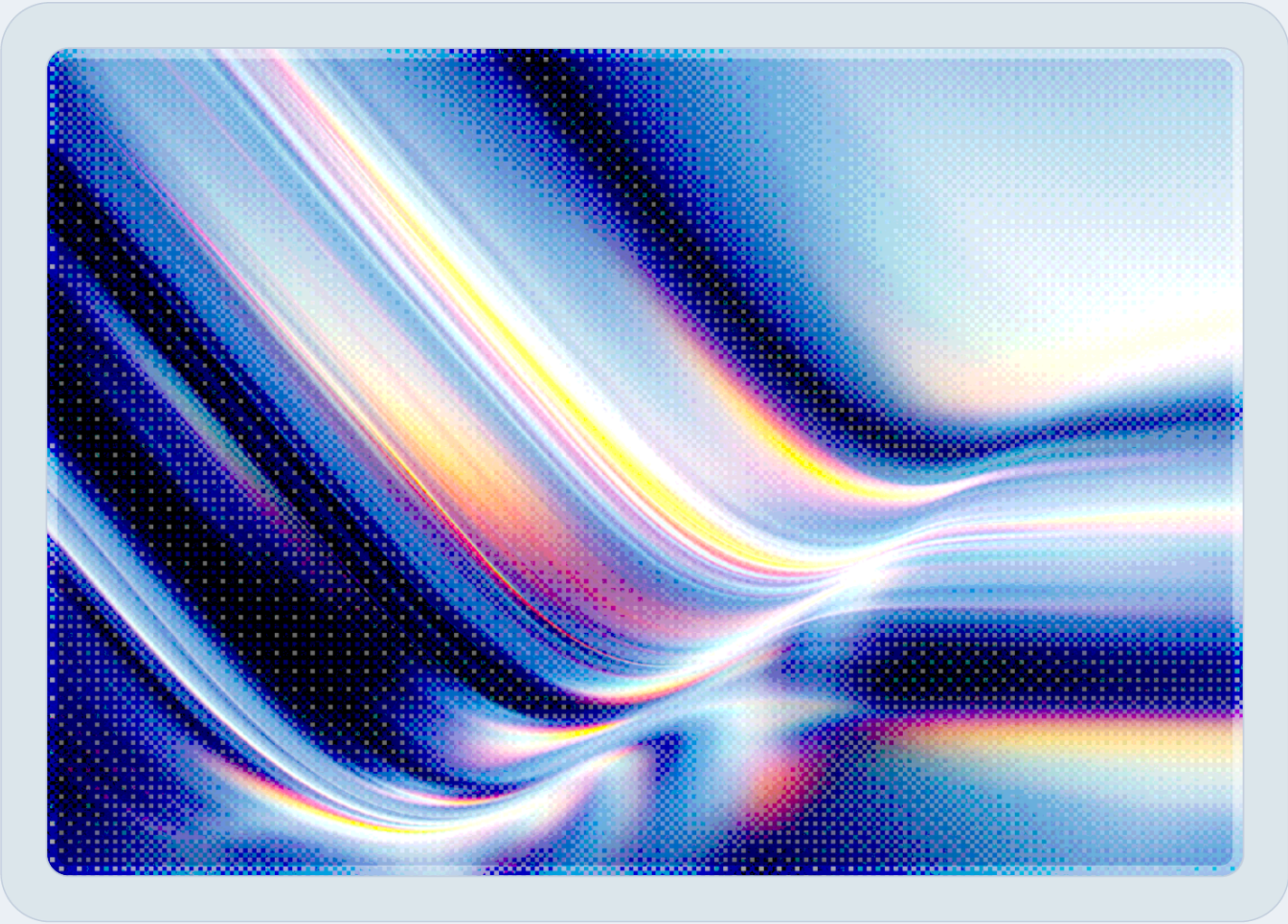
CSAI Foundation | Cloud Security Alliance

npm Supply Chain Under Siege: TeamPCP, Miasma, and npm v12

Defending the JavaScript Ecosystem Against Credential-Stealing Worms and CI/CD Pipeline Compromise

2026-06-11

AI-assisted Rapid Research



CSAI

cloud security alliance®

Table of Contents

Executive Summary 4

1. Introduction: npm as Critical Developer Infrastructure 5

2. The TeamPCP Threat Cluster: A Coordinated, Phased Campaign 6

- 2.1 Origins and the CanisterWorm Operation (March 2026)
- 2.2 Targeting the AI/ML Toolchain: The Trivy-to-LiteLLM Kill Chain
- 2.3 The TanStack Compromise: Defeating SLSA Build Level 3
- 2.4 Mini Shai-Hulud Goes Open Source: A Strategic Escalation

3. Miasma: The Red Hat Campaign and Its Aftermath 10

- 3.1 The Attack Vector and Technical Execution
- 3.2 Worm Behavior and Self-Propagation
- 3.3 Signed Attestations as False Assurance
- 3.4 Attribution and Copycat Risk

4. The SLSA Provenance Problem: What Attestations Can and Cannot Guarantee 13

5. npm v12: GitHub's Structural Security Response 14

- 5.1 Install Scripts Disabled by Default
- 5.2 Blocking Git and Remote URL Dependencies
- 5.3 Preparing for the npm v12 Transition
- 5.4 What npm v12 Cannot Fix

6. The AI Toolchain as an Elevated-Risk Attack Surface 17

7. Recommendations 18

- 7.1 Immediate Actions
- 7.2 Medium-Term Program Investments
- 7.3 Strategic Posture

8. CSA Resource Alignment 20

9. Conclusions 21

References 22

Executive Summary

The npm package registry serves as foundational infrastructure for modern software development. With more than two million packages and billions of weekly downloads, it underpins web applications, cloud services, developer tooling, and an increasing proportion of artificial intelligence and machine learning infrastructure. That centrality makes npm an extraordinarily attractive target: a single compromised package can reach millions of downstream developers and the production systems they build.

Between March and June 2026, a threat cluster designated TeamPCP executed a multi-phase supply chain campaign of unprecedented scope and sophistication against the npm ecosystem. The campaign began with CanisterWorm, moved through the deliberate targeting of AI/ML tooling—notably the LiteLLM unified LLM gateway—and reached a technical apex with the TanStack compromise in May 2026, which became the first documented case of a malicious npm package carrying cryptographically valid SLSA Build Level 3 provenance attestations. TeamPCP subsequently open-sourced their Mini Shai-Hulud worm framework on GitHub under a permissive license, effectively democratizing their attack capabilities. Within weeks, the Miasma campaign — using a worm framework consistent with the publicly released Mini Shai-Hulud codebase — compromised 32 packages under Red Hat's `@redhat-cloud-services` npm scope, affecting more than 600,000 monthly downloads, though attribution to TeamPCP specifically remains a vendor assessment rather than a confirmed designation.

On June 9, 2026, GitHub announced upcoming breaking changes for npm v12, scheduled for release in July 2026, that directly constrain the execution vectors these campaigns have exploited. npm v12 will disable install script execution by default, block resolution of Git-sourced and remote-URL dependencies, and require explicit project-level allowlisting for any of these behaviors. These changes are available behind warnings in npm 11.16.0 and newer, enabling organizations to audit and prepare before the mandatory transition. GitHub describes these changes as what it considers the most significant structural security overhaul in the package manager's history.

This whitepaper provides security teams, platform engineers, and security architects with a comprehensive account of the threat landscape, an analysis of npm v12's defensive coverage and its limits, and a prioritized set of recommendations anchored in CSA's existing frameworks for supply chain security and AI governance.

1. Introduction: npm as Critical Developer Infrastructure

The npm registry's role in software supply chains has expanded far beyond its origins as a JavaScript package manager. It now serves as the distribution layer for TypeScript libraries, build tooling, test frameworks, security scanners, IDE extensions, and the developer-facing SDKs for all major cloud providers and AI platforms. Organizations building AI-enabled products rely on npm not only for frontend and backend dependencies but increasingly for the orchestration layers, agent frameworks, and LLM client libraries that connect their systems to foundation models. This expanded scope means that a compromise of a widely downloaded npm package can cascade across domains that were not historically considered part of the JavaScript supply chain.

The trust model that underlies npm dependency management was designed for a different era. When a developer runs `npm install`, the package manager fetches declared dependencies, resolves their transitive graphs, and—until npm v12—executes any lifecycle scripts those packages define. This execution happens with the privileges of the installing user, in an environment that typically includes cloud credentials, SSH keys, CI/CD tokens, and access to the broader development workstation or runner environment. Security controls like Sigstore-based provenance attestations and SLSA framework certification have emerged in recent years to bring transparency and integrity assurances to this process, but as the events of 2026 have demonstrated, these controls operate on a fundamentally different threat model than the one TeamPCP has exploited.

The sustained nature of the TeamPCP campaign—spanning multiple ecosystems, targeting security tooling itself, and adapting across successive waves of remediation—marks a shift in the supply chain threat landscape. While many prior npm compromises were opportunistic—a single package, a single actor, limited propagation—the TeamPCP campaign is characterized by strategic objective sequencing, automated self-propagation, and a deliberate focus on the credentials that enable further ecosystem-level compromise rather than on the end-user systems downstream packages are deployed to. Understanding this campaign's structure is prerequisite to building defenses that address root causes rather than surface symptoms.

2. The TeamPCP Threat Cluster: A Coordinated, Phased Campaign

2.1 Origins and the CanisterWorm Operation (March 2026)

The TeamPCP campaign became publicly visible on March 20, 2026, when researchers identified CanisterWorm, a self-propagating npm worm that used stolen npm authentication tokens to compromise more than sixty packages in a single automated wave [17]. The pattern established by CanisterWorm defined the operational template for all subsequent TeamPCP activity: harvest credentials from one entry point, use those credentials to infect additional packages, embed a payload that repeats the cycle on the next developer or system that installs the infected package, and exfiltrate the collected material to attacker-controlled infrastructure.

CanisterWorm's self-propagation mechanism exploited the npm lifecycle script system in a straightforward way. Infected packages inserted malicious code into their `postinstall` hooks, which executed when any downstream developer ran `npm install`. The payload searched the host environment for npm authentication tokens, typically stored in `.npmrc` files or available as environment variables in CI/CD contexts, and used them to publish new malicious versions of other packages the victim maintained. This worm-like behavior meant that each new victim became an unwitting propagation node, extending the campaign's reach without requiring additional attacker involvement.

The scale of the initial CanisterWorm wave—sixty-plus packages in a single operation—indicated that TeamPCP had either accumulated a significant portfolio of stolen npm credentials before the campaign began or had automated the credential-to-publish pipeline to operate at machine speed. Subsequent analysis by Picus Security confirmed the latter: the operation was largely automated, with the worm's credential-harvesting and republication cycle executing in minutes after an initial installation [17].

2.2 Targeting the AI/ML Toolchain: The Trivy-to-LiteLLM Kill Chain

The most strategically significant phase of the TeamPCP campaign ran from March 19 through March 27, 2026, and revealed what appears to be a deliberate intent to harvest credentials for AI service providers alongside the conventional developer secrets that supply chain attacks had previously targeted [31].

The kill chain began with the compromise of Trivy, an open-source vulnerability scanner developed by Aqua Security and widely used in CI/CD pipelines for container image and dependency scanning. By poisoning Trivy's GitHub Action in mid-March 2026, TeamPCP positioned their payload inside the security scanning

step of thousands of pipelines—a position that, by definition, runs with elevated access to the secrets those pipelines protect. Trivy's role as a security tool meant that its GitHub Action was trusted and that the credentials available to it included the publishing tokens for the packages it was used to audit. Researchers at Datadog Security Labs and Snyk traced the LiteLLM compromise directly to this Trivy vector [18, 19].

On March 24, 2026, two malicious releases of LiteLLM—versions 1.82.7 and 1.82.8—appeared on PyPI, published using the maintainer's PyPI credentials that had been exfiltrated through the compromised Trivy action [18]. LiteLLM is a unified gateway library that routes requests to more than one hundred large language model APIs, including those of OpenAI, Anthropic, Google, Mistral, and dozens of others. With approximately 3.4 million daily downloads, it sits at the center of many AI application architectures, and—critically—it typically runs with direct access to the API keys for every LLM provider an organization uses [34]. The malicious LiteLLM releases deployed a three-stage payload: credential harvesting across cloud platforms, Kubernetes lateral movement, and a persistent backdoor for remote code execution. The malicious versions remained available on PyPI for approximately three hours before the package was quarantined [18].

The deliberate targeting of LiteLLM, rather than a generic infrastructure package, indicates that TeamPCP had mapped the AI developer toolchain's access patterns before executing this phase. An organization whose CI/CD pipeline runs LiteLLM likely has API keys for every foundation model it calls, meaning a single successful compromise could yield credentials across multiple AI providers simultaneously. This framing—the AI toolchain as a high-value credential aggregator—distinguishes the March 2026 phase from conventional supply chain attacks and has direct implications for how organizations should scope their software supply chain risk assessments.

The same campaign wave also reached Checkmarx KICS (Keeping Infrastructure as Code Secure), another widely-used security tool, further reinforcing the pattern of targeting the security-tooling layer as a privileged position within CI/CD pipelines [18].

2.3 The TanStack Compromise: Defeating SLSA Build Level 3

The most technically consequential event in the TeamPCP campaign occurred on May 11, 2026, when attackers published 84 malicious npm package artifacts across 42 packages in the `@tanstack` namespace between 19:20 and 19:26 UTC [14]. TanStack's libraries—including TanStack Query, TanStack Router, and TanStack Table—are widely used across the JavaScript ecosystem for state management, routing, and data display. The six-minute publication window and the method of compromise marked this incident as a fundamental challenge to the supply chain security controls the community had invested heavily in building.

The attack chain combined three vulnerabilities in TanStack's CI/CD pipeline in sequence. First, attackers exploited the `pull_request_target` workflow trigger—a well-known pattern sometimes called "Pwn Request"—which grants workflows from forked repositories access to repository secrets while running in the context of the base repository. Second, they poisoned the GitHub Actions workflow cache across the fork-to-base trust boundary, injecting attacker-controlled code into the build environment. Third, they extracted an OIDC token from the GitHub Actions runner process at runtime, using the legitimate OIDC token to authenticate through TanStack's trusted publish workflow and obtain npm publish rights [14, 33].

What made this compromise particularly significant was its implications for SLSA provenance. The Sigstore attestations on the malicious `@tanstack/*` versions are cryptographically valid: they accurately record that the packages were built and published by TanStack's release workflow, running on `refs/heads/main` in the TanStack/router repository. The SLSA provenance was not forged—it is a truthful attestation of what actually happened in the build system. The malicious packages were published through TanStack's legitimate release pipeline, using its trusted OIDC identity, after attacker-controlled code hijacked the runner mid-workflow [12, 23]. This was the first documented case of a malicious npm package carrying valid SLSA Build Level 3 provenance attestations, and it required the security community to revisit its assumptions about what provenance attestations do and do not guarantee.

The broader impact of the May 11–19 wave extended well beyond the TanStack packages themselves. TeamPCP used stolen credentials to poison more than 170 npm and PyPI packages, hijack a VS Code extension with 2.2 million installs, and exfiltrate a significant number of internal GitHub repositories — one account reported approximately 3,800 repositories [26], though other technical analyses place the figure at 2,500 or more [39]; the precise count has not been confirmed by primary parties. Mistral AI and UiPath were among the organizations affected. The AntV data visualization ecosystem was struck on May 19, 2026, with 637 malicious package versions across 323 packages published in two rapid bursts over approximately 27 minutes, demonstrating the automated, high-velocity execution capability the operation had developed by this point in the campaign [21].

2.4 Mini Shai-Hulud Goes Open Source: A Strategic Escalation

On May 12, 2026—the day after the TanStack compromise became public—TeamPCP published the source code for the Mini Shai-Hulud worm framework on GitHub under an MIT license, with the release message: "Shai-Hulud: Open Sourcing The Carnage" [16]. The framework is a modular TypeScript and Bun toolkit implementing credential harvesting, supply chain poisoning, and encrypted data exfiltration. It includes collectors targeting CI/CD environments, cloud identity providers, developer workstations, and the GitHub Actions runner memory-scraping technique used in the TanStack compromise.

The decision to open-source the framework is best understood as a strategic move rather than a technical one. By releasing the tooling publicly, TeamPCP lowered the barrier for any actor with moderate technical capability to replicate or adapt their attack patterns. The framework's open availability means that

attribution of subsequent campaigns to TeamPCP becomes genuinely difficult: similar TTPs may indicate continued TeamPCP activity, adoption by another actor, or independent rediscovery of the same techniques. This attribution complexity was demonstrated almost immediately by the Miasma campaign, discussed in the following section [3, 5].

3. Miasma: The Red Hat Campaign and Its Aftermath

3.1 The Attack Vector and Technical Execution

On June 1, 2026, researchers identified 32 malicious package releases across the `@redhat-cloud-services` npm scope, collectively spanning more than 90 package versions [3, 7]. The `@redhat-cloud-services` scope is maintained by the RedHatInsights project and encompasses JavaScript client libraries used by Red Hat's cloud management console, CI/CD tooling, and internal developer tooling. According to OX Security, the affected packages collectively had approximately 600,000 monthly downloads at the time of compromise [6].

The compromise originated in the upstream `RedHatInsights/javascript-clients` repository, where a Red Hat employee's GitHub account was used to push orphan commits directly into several repositories, bypassing standard branch protection and code review requirements [4]. The compromised account's push access triggered the repositories' GitHub Actions workflows, which were configured with `id-token: write` permission and designed to request OIDC tokens for authenticated npm publishing. Because the attacker controlled the code executed by these workflows, they were able to use the OIDC token exchange mechanism to obtain npm publish rights and release malicious versions of the affected packages with all the signatures that legitimate releases would carry [4].

The malicious payload was delivered through npm's `preinstall` lifecycle hook—the first script to run during `npm install`—and consisted of a heavily obfuscated 4.29 megabyte dropper script [5]. On developer workstations, the dropper harvested SSH keys, browser stored credentials, wallet files, and CLI authentication tokens for cloud providers. In CI/CD environments, it performed runner memory scraping to extract GitHub Actions secrets, escalated privileges where passwordless sudo was configured, and used stolen npm OIDC tokens to republish poisoned versions of other packages the affected organization maintained. The worm's self-propagation capability meant that the Red Hat packages were not merely a delivery vehicle but a propagation node: any downstream developer or pipeline that installed one of the compromised packages could become the next origin point for further infection.

Particularly significant were the credential collectors targeting cloud identities. Analysis by Wiz and other researchers found specialized modules for collecting GCP service account tokens, Azure managed identity credentials, and AWS instance metadata, reflecting the cloud-focused deployment context of the `@redhat-cloud-services` audience [3]. This targeting precision—a credential harvester tailored to the likely infrastructure of a package's downstream consumers—indicates either detailed prior reconnaissance or a modular framework designed to activate relevant collectors based on detected environment context.

Malicious versions were published on June 1, 2026; the compromise was publicly disclosed approximately at 1 PM UTC, and most malicious versions were revoked by 2 PM UTC, representing a response window of approximately one hour from public disclosure [4]. The speed of remediation reflected coordinated effort by Red Hat's security team and GitHub's platform team, though the window was sufficient for any automated pipeline running on the affected scope to have installed and executed the payload.

3.2 Worm Behavior and Self-Propagation

The self-propagating design of the Miasma payload follows the same architectural pattern as TeamPCP's earlier Mini Shai-Hulud campaigns but adapted for the specific publishing permissions available through Red Hat's OIDC-based release workflows. Once the initial preinstall dropper executed and harvested npm publishing credentials, the worm component used those credentials to republish poisoned versions of any packages the victim maintained or had publish rights to. This behavior is distinct from conventional supply chain attacks, where a compromise is typically limited to the initially affected package family: the worm's propagation mechanism means that containment requires identifying and revoking every credential set the initial payload reached, not just the originally compromised packages.

The worm's propagation also highlights a systemic risk in how npm package publishing credentials are managed in CI/CD environments. When a GitHub Actions runner has `id-token: write` permission and the corresponding npm publishing workflow is configured for OIDC-based trusted publishing, any code that executes on that runner—including code injected through workflow triggers or cache poisoning—can obtain valid npm publish tokens. The OIDC token is a narrow-scope, short-lived credential by design, but that scope includes the ability to publish all versions of all packages the associated account maintains. In organizations with large npm package portfolios, this represents a significant blast radius from a single runner compromise.

3.3 Signed Attestations as False Assurance

The Miasma compromise reinforces the lesson established by the TanStack attack: cryptographic attestations verify the provenance chain, not the integrity of the code that runs inside it. Because the malicious Miasma releases were published through the legitimate RedHatInsights OIDC publishing workflow—using real tokens obtained via a genuine workflow invocation—the resulting Sigstore signatures and SLSA provenance attestations are cryptographically valid. A downstream consumer checking npm provenance before installation would observe that the package was built and published by a trusted Red Hat workflow, on the expected repository, from a recent commit. None of these attestations are false. The malicious code was genuinely built and published through those legitimate channels [4, 23].

This is a structural limitation of current supply chain attestation frameworks, and the SLSA project has acknowledged it explicitly in their analysis of the Mini Shai-Hulud campaign [23]. SLSA Build Level 3 provides strong guarantees that the build platform's infrastructure was not tampered with and that the build ran in an isolated, ephemeral environment. It does not—and was not designed to—guarantee that the code checked into the repository was itself free of malicious content, or that the credentials used to trigger the build were legitimately held by their rightful owner. The threat model that SLSA addresses is infrastructure-level tampering, not credential theft and account compromise.

Mondoo's analysis of the 2026 npm threat landscape captures this concisely: signed attestations tell you that a package was built where it claims to have been built; they do not tell you that the person who pushed the build trigger was who they claimed to be, or that their account had not been compromised [24]. This distinction matters enormously for organizational security posture because it means that provenance verification, while valuable, cannot substitute for the controls that govern who can push code to a repository and who can trigger publishing workflows.

3.4 Attribution and Copycat Risk

Vendor reporting on the Miasma campaign attributes the activity to the TeamPCP cluster, based on the observed use of Mini Shai-Hulud framework components and shared technical indicators with prior TeamPCP operations [5, 9]. That attribution is plausible and the technical similarity is well-documented. However, because TeamPCP open-sourced the Mini Shai-Hulud framework on May 12, 2026—under a permissive license, with modular code that can be adapted without deep expertise—any actor seeking to conduct npm supply chain attacks has had access to the same techniques and tooling for three weeks prior to the Miasma campaign. Vendor researchers have noted this explicitly: the observed similarities should be treated as evidence of TTP overlap rather than definitive attribution, and organizations should plan their defenses accordingly regardless of which actor is responsible [9].

The copycat dimension of the Mini Shai-Hulud open-source release has significant long-term implications. Before the release, npm self-propagating worms required meaningful development effort and ecosystem-specific knowledge. After the release, the capability is available to any actor who can operate TypeScript tooling and navigate npm's credential-exchange interfaces. The open-sourcing of Shai-Hulud represents a capability proliferation event for supply chain attacks in the same sense that the public release of prior exploit frameworks democratized other attack categories.

4. The SLSA Provenance Problem: What Attestations Can and Cannot Guarantee

The TanStack and Miasma incidents together constitute a natural experiment in the limits of SLSA-based provenance assurance, and their implications deserve separate treatment before discussing mitigations.

SLSA provenance attestations were designed to answer a specific question: was this package built by the system it claims was built by, without tampering of the build infrastructure itself? At Build Level 3, the framework additionally requires that the build ran in an ephemeral, isolated environment, making it harder for a persistent attacker inside the build system to modify build outputs without detection. These are meaningful guarantees that protect against a specific class of build-system compromises, and they represent a genuine improvement in the supply chain security posture of packages that have earned them.

What SLSA does not guarantee—and was not designed to guarantee—is that the code committed to the repository was free of malicious intent, or that the identity that triggered the build had not been compromised. The TanStack attacker used cache poisoning to inject malicious code into the build environment at runtime, after the SLSA isolation boundary but before the artifacts were sealed. The Miasma attacker used a compromised employee account to push commits directly to the repository, which then triggered the legitimate publishing workflow. In both cases, the attestation accurately describes what happened—a truthful record of a build that produced malicious artifacts.

The SLSA project's response to the Mini Shai-Hulud analysis recommends a layered approach: SLSA attestations should be combined with branch protection policies that require multi-party review of commits before they can trigger publishing workflows, runtime policy enforcement that detects anomalous behavior in build environments, and developer identity verification strong enough to make account compromise significantly harder [23]. These layers address the gap between what provenance attestations verify and what organizations actually need to know.

This framing—provenance as one layer in a defense-in-depth stack, not a complete control—is important for organizations that have invested in npm provenance as a supply chain security measure. Provenance verification continues to provide value, particularly against build-system tampering and artifact substitution attacks. It should not, however, be presented to stakeholders as protection against account compromise or repository-level code injection.

5. npm v12: GitHub's Structural Security Response

On June 9, 2026, GitHub announced the upcoming breaking changes for npm v12, estimated for release in July 2026, representing what GitHub describes as the most significant security-focused overhaul of the package manager's default behavior since its inception [1]. The changes directly address the execution vectors that TeamPCP and Miasma have exploited most effectively.

5.1 Install Scripts Disabled by Default

The centerpiece of npm v12's security changes is the disabling of install script execution by default. Under current behavior, when a package declares `preinstall`, `install`, or `postinstall` lifecycle scripts, npm executes them automatically during installation with the privileges of the running process. This behavior is the mechanism both TeamPCP and Miasma exploited to run credential-harvesting payloads the moment a compromised package was installed.

Under npm v12, install scripts from dependencies will not execute unless the project has explicitly allowlisted them [1, 38]. This allowlisting is maintained in `package.json` and committed to version control, making it visible in code review and auditable over time. The allowlist applies at the project level rather than the package level, meaning that transitive dependencies—a common attack surface in prior supply chain compromises—must also be explicitly approved. Notably, the restriction applies to native build dependencies as well: packages with a `binding.gyp` file that trigger an implicit `node-gyp rebuild` are also blocked unless explicitly approved, closing a bypass path where a package could trigger code execution without declaring a conventional install script [8].

The `npm approve-scripts` command, available in npm 11.16.0 and newer, allows projects to audit which of their dependencies have install scripts and selectively approve the ones they have reviewed. The command outputs a list of pending script approvals, provides information about the packages and scripts involved, and updates `package.json` when approvals are committed. GitHub recommends that teams upgrade to npm 11.16.0 or newer now, run their normal install workflow, review the generated warnings, and use `npm approve-scripts --allow-scripts-pending` to build their allowlist before the v12 transition [1].

5.2 Blocking Git and Remote URL Dependencies

npm v12 also disables resolution of Git-sourced and remote-URL dependencies by default. Under current behavior, a dependency declared as a Git repository URL or a remote tarball URL is resolved and installed without additional prompting, even though these dependency types execute code from sources outside the npm registry's security review and malware scanning infrastructure. The `--allow-git` and `--allow-remote` flags, new in v12, must be explicitly invoked to permit these dependency types, and their use requires conscious review [1].

The Git dependency restriction addresses a specific attack vector that predates TeamPCP: a Git dependency's repository can contain an `.npmrc` file that overrides the Git executable path, enabling code execution even when `--ignore-scripts` is set. This vector requires the attacker to control the Git repository being referenced, but given the prevalence of Git URL dependencies in monorepo tooling and private packages, it represents a meaningful attack surface that the v12 default closes.

5.3 Preparing for the npm v12 Transition

All three security changes are available behind warning mode in npm 11.16.0 and newer, meaning organizations can begin their audit-and-allowlist process immediately without waiting for the v12 release. The recommended preparation workflow is to upgrade to npm 11.16.0 or later, run the standard install process against the existing dependency tree, collect the resulting warnings about install scripts and non-registry dependencies, use `npm approve-scripts` to review and approve only those scripts that have been examined and deemed necessary, and commit the resulting `package.json` changes to version control [1, 2]. This process surfaces the organization's actual install script exposure—often a surprising quantity of transitive dependency scripts that teams were unaware were executing—and forces a deliberate decision about each one.

The transition will create friction for projects that rely on native module builds, particularly those using `node-gyp` for compiled binaries. Maintainers of these packages are encouraged to declare their build scripts explicitly and communicate their presence to downstream consumers. The npm team has indicated that the approve-scripts workflow is designed to make this a one-time cost for stable dependency trees: once a project's allowlist is established and committed, subsequent installs against the same dependency versions will not require re-approval.

5.4 What npm v12 Cannot Fix

The npm v12 changes directly address the install-time execution vector and close significant attack surface. They do not, however, address the full attack surface that the 2026 campaign exploited. The changes do not prevent a malicious package from containing malicious code in its primary module that executes when

imported rather than at install time. They do not address the account compromise and CI/CD pipeline hijacking vectors that allowed attackers to publish malicious versions of legitimate packages in the first place. They do not provide runtime detection of credential harvesting or network exfiltration behavior. And they do not prevent a developer who has explicitly approved a package's install script from later receiving a malicious update to that same package without re-review.

These limitations are not criticisms of the npm v12 design—they reflect the inherent scope of a package manager. npm can control what executes during installation; it cannot substitute for CI/CD security controls, identity and access management hygiene, or runtime behavior monitoring. The responsible framing for npm v12 is that it removes one significant class of attack vector that has been heavily exploited, making the ecosystem's trust model substantially more resilient, while leaving intact the need for the surrounding layers of security control.

6. The AI Toolchain as an Elevated-Risk Attack Surface

The deliberate targeting of AI/ML developer tooling—LiteLLM, Trivy, Checkmarx KICS, and AI-adjacent packages—in the March 2026 phase of the TeamPCP campaign reflects a threat actor insight that deserves attention from security architects and AI governance teams: the AI developer toolchain aggregates an unusually high density of sensitive credentials, and the affected organizations demonstrate that existing supply chain security programs had not yet extended to cover AI-specific toolchain dependencies with the same rigor applied to conventional infrastructure packages.

LiteLLM's role as a unified LLM API gateway is representative of this dynamic. An organization that deploys LiteLLM in its CI/CD pipeline or application runtime has, by necessity, configured it with API keys for every LLM provider it calls. A single successful compromise of the LiteLLM package reaches all of those credentials simultaneously. The same pattern applies to multi-model orchestration frameworks, AI agent harnesses, and the testing infrastructure used to evaluate model outputs. Unlike conventional infrastructure packages, whose compromise might yield database credentials or cloud access tokens, AI toolchain packages often aggregate credentials across multiple external providers in a single process.

Beyond credential aggregation, AI agent frameworks and orchestration layers present additional supply chain risks that are not present in conventional software. An AI agent that processes external content—web pages, documents, user messages—and calls tools in response creates an execution surface where adversarial content in the environment can influence what code is executed. Supply chain attacks that target the tools an agent can invoke, or the prompts embedded in agent skill packages, represent an attack vector that is qualitatively different from conventional install-time payload delivery; the CSA MAESTRO framework explicitly addresses this threat class at layer 5 (agentic supply chain risks). Organizations building AI-powered products should apply the same supply chain rigor to AI-specific dependencies—model client libraries, orchestration frameworks, evaluation tooling—that they apply to infrastructure dependencies, and should additionally audit what credentials each layer of the AI stack has access to and whether that access is scoped to the minimum necessary.

7. Recommendations

7.1 Immediate Actions

Organizations should treat the following actions as time-sensitive given the active threat campaign and the July 2026 npm v12 release window.

Upgrading to npm 11.16.0 or newer and running `npm approve-scripts --allow-scripts-pending` across all active repositories is the most direct preparation step available. The output of this audit will surface transitive install scripts that may have been executing unnoticed in current pipelines and enable a deliberate review before the v12 transition. Projects that have not yet conducted this audit may be running install scripts from a large number of transitive dependencies without awareness.

Reviewing CI/CD workflow permissions for `id-token: write` scope is urgently warranted in light of the TanStack and Miasma attacks. Any workflow that requests an OIDC token and exchanges it for npm publish rights should be audited for the `pull_request_target` misconfiguration pattern and for external pull requests or forks that can trigger it. GitHub's documentation on `pull_request_target` security provides current guidance; workflows that are not explicitly designed to handle fork pull requests should either not use `pull_request_target` or should add explicit protections against fork-triggered execution.

Rotating npm authentication tokens for any account or package that may have been exposed to the Miasma campaign is appropriate for organizations using `@redhat-cloud-services` packages and for any organization whose CI/CD pipelines might have installed affected packages during the June 1, 2026 window before revocation. Because the Miasma worm used stolen npm credentials to republish additional packages, the blast radius of the compromise may extend to any package maintained by developers whose workstations or pipelines executed the payload.

7.2 Medium-Term Program Investments

Establishing an approved-install-script inventory as a formal security artifact—reviewed in the same change management process as dependency updates—aligns with the intent of npm v12's allowlisting model and extends it into an ongoing governance practice rather than a one-time migration activity. The inventory should capture the rationale for each approved script, the version range to which approval applies, and the reviewer who approved it.

Implementing branch protection rules that require multi-party code review before commits can trigger publishing workflows addresses the repository-level injection vector the Miasma attacker exploited. The SLSA project's guidance recommends combining provenance attestations with branch protection policies that prevent any single individual from committing code and triggering a publish without review [23]. This combination does not make attestations redundant; it addresses the gap that attestations alone leave open.

Conducting supply chain audits specifically for AI toolchain dependencies—inventorying every package in the AI/ML dependency graph that has access to LLM API keys, cloud credentials, or model service tokens—provides visibility into the credential aggregation risk that the LiteLLM compromise illustrated. The audit output should feed into a least-privilege review: does each component in the AI stack actually require access to every credential currently available to it, or can the access surface be narrowed?

Extending SBOM generation and tracking to cover AI-specific dependencies is consistent with the CISA 2025 Minimum Elements guidance for software bills of materials, which now accommodates AI component enumeration [28], and with the CISA and G7 joint guidance on Software Bill of Materials for AI, which provides minimum elements specifically for AI developers and deployers [29]. Organizations that already maintain SBOMs for their conventional software supply chains should explicitly include AI SDK and framework dependencies within scope.

7.3 Strategic Posture

At a strategic level, the 2026 npm campaign illustrates that supply chain security is not a problem that can be solved by any single control layer. The install script execution vector has been available for years and is now being closed by npm v12; the SLSA provenance framework provides genuine value and has genuine limits; and both the npm registry's malware scanning and GitHub's security infrastructure responded to the Miasma and TanStack incidents faster than many organizations' own detection capabilities would have. The remaining risk surface lies in the controls that prevent account compromise, detect anomalous CI/CD behavior, and ensure that the credential scope available to any given pipeline step is proportionate to its actual requirements.

Organizations whose security programs have not yet assessed software supply chain risk as a distinct discipline—separate from conventional vulnerability management and application security—should treat the 2026 campaign as a forcing function for that assessment. The CSA STAR program, the AICM framework's supply chain control domains, and the CCM's STA control family provide structured starting points for this assessment and for communicating supply chain security posture to auditors and enterprise customers.

8. CSA Resource Alignment

Several CSA frameworks and publications provide directly applicable guidance for the threats described in this whitepaper.

MAESTRO (Agentic AI Threat Modeling) addresses supply chain security as a distinct threat category for agentic AI systems at threat layer 5. The MAESTRO framework recognizes that AI agents that invoke tools and external skills create a runtime attack surface where compromised tool packages or skill repositories can influence agent behavior beyond their intended scope. Organizations deploying AI agent infrastructure should apply MAESTRO's layer 5 analysis to their tool and skill dependency graphs, not only to the model and orchestration layers.

CSA AI Controls Matrix (AICM) includes controls within the AI Supply Chain Management domain that address third-party component security, software integrity verification, and the governance of AI-specific dependencies. The AICM is explicitly a superset of CCM and should be the default framework reference for AI-enabled product supply chains rather than the CCM alone. The events of 2026 reinforce the importance of AICM's supply chain controls, particularly those addressing publishing credential management and CI/CD pipeline security for AI component releases.

Software Transparency: Securing the Digital Supply Chain (CSA Publication) provides foundational guidance on SBOM practices, open-source risk management, and CI/CD pipeline integrity that applies directly to the npm ecosystem threat scenarios described here. Organizations that have not yet adopted the practices described in this publication should treat the 2026 campaign as confirmation of the threat model it describes.

CCM STA (Supply Chain Management, Transparency, and Accountability) Control Family provides the governance framework for third-party risk management that encompasses npm dependencies and the CI/CD tools used to build, test, and publish software. STA controls for vendor risk assessment, software integrity verification, and supply chain incident response map directly to the organizational capabilities that proved decisive in the Miasma response.

CSA Zero Trust Guidance informs the network and identity controls that bear on CI/CD pipeline security. The principle that no runtime component—including a build runner—should be trusted implicitly, and that OIDC token scopes should be constrained to the minimum necessary for each workflow step, is a direct application of Zero Trust to the CI/CD context that the TanStack and Miasma attacks exploited.

CSA Labs has published research notes on both the Miasma campaign [30] and the TeamPCP AI toolchain attack [31] that provide additional technical depth on individual incidents and serve as companion references to this whitepaper's broader campaign analysis.

9. Conclusions

The npm ecosystem has undergone a period of sustained, sophisticated attack that challenges assumptions embedded in both the trust model of open-source package management and the security controls organizations have relied upon to verify software supply chain integrity. The TeamPCP campaign—spanning CanisterWorm, the deliberate targeting of AI/ML tooling, the TanStack SLSA Build Level 3 defeat, and the open-sourcing of Mini Shai-Hulud—represents a qualitative shift in the documented capabilities and strategic ambition of npm-targeting threat actors. The Miasma campaign against Red Hat's npm packages demonstrated that the open-sourced framework has immediately lowered barriers for additional actors, and that the attribution challenge this creates is itself a strategic asset for attackers.

GitHub's npm v12 security overhaul directly addresses the install-script execution vector that has been central to self-propagating worm behavior in the npm ecosystem. Organizations that prepare for the v12 transition now—by upgrading to npm 11.16.0, auditing their install script exposure, and establishing allowlists through the `npm approve-scripts` workflow—will close one of the most significant attack surfaces that has been exploited in the 2026 campaign. The transition requires investment but is not optional: organizations that do not manage it actively will find their existing pipelines broken when v12 becomes the default, creating pressure to grant broad script allowances without adequate review.

At the same time, npm v12 does not substitute for the identity and access management controls that prevent account compromise, the CI/CD pipeline security controls that prevent runtime code injection, or the runtime monitoring capabilities that detect credential exfiltration after it occurs. The lesson of the TanStack and Miasma incidents is that cryptographic attestations and SLSA provenance provide genuine value within a bounded threat model and must be combined with controls that address the threat model gaps they do not cover.

The deliberate targeting of AI toolchain packages—and the unique credential aggregation risk they present—should prompt every organization building AI-enabled products to extend their software supply chain security program explicitly to cover AI-specific dependencies. The LiteLLM compromise illustrated the consequences of treating AI framework packages as lower-risk than conventional infrastructure dependencies. They are not. Organizations whose AI toolchain includes a unified LLM gateway may find that a single package has access to more external service credentials than any other component in their stack. The AI toolchain should be managed accordingly.

References

[1] GitHub. "Upcoming Breaking Changes for npm v12." GitHub Changelog, June 9, 2026.

[2] BleepingComputer. "GitHub Announces npm Security Changes to Tackle Supply-Chain Attacks." BleepingComputer, 2026.

[3] Wiz Research. "Miasma: Supply Chain Attack Targeting RedHat npm Packages." Wiz Blog, 2026.

[4] Microsoft Security Response Center. "Preinstall to Persistence: Inside the Red Hat npm Miasma Credential-Stealing Campaign." Microsoft Security Blog, June 2, 2026.

[5] Snyk. "Miasma Attack Hits Red Hat npm Packages." Snyk Security Blog, 2026.

[6] OX Security. "600,000 Monthly Downloads Affected: Miasma Supply Chain Attack Is Back on npm." OX Security Blog, 2026.

[7] The Hacker News. "Miasma Supply Chain Attack Compromises Red Hat npm Packages with Credential-Stealing Worm." The Hacker News, 2026.

[8] StepSecurity. "Miasma npm Supply Chain Attack: Self-Spreading Worm via Phantom Gyp." StepSecurity Blog, 2026.

[9] Palo Alto Networks Unit 42. "The npm Threat Landscape: Attack Surface and Mitigations (Updated June 2)." Unit 42 Blog, 2026.

[10] Wiz. "Mini Shai-Hulud Strikes Again: TanStack + More npm Packages Compromised." Wiz Blog, 2026.

[11] Upwind. "Miasma: A Worming npm Supply Chain Attack on Red Hat Cloud Services." Upwind Blog, 2026.

[12] StepSecurity. "TeamPCP's Mini Shai-Hulud Is Back: A Self-Spreading Supply Chain Attack Compromises TanStack npm Packages." StepSecurity Blog, 2026.

[13] Orca Security. "TanStack and 160+ npm/PyPI Packages Compromised in Supply Chain Worm Attack." Orca Security Blog, 2026.

[14] TanStack. "Postmortem: TanStack npm Supply-Chain Compromise." TanStack Blog, 2026.

[15] TanStack. "Hardening TanStack After the npm Compromise." TanStack Blog, 2026.

[16] The Register. "Malware Crew TeamPCP Open-Sources Its Shai-Hulud Worm on GitHub." The Register, May 13, 2026.

[17] Picus Security. "CanisterWorm: How TeamPCP Turned the npm Ecosystem Into a Weapon." Picus Security Blog, 2026.

[18] Datadog Security Labs. "LiteLLM and Telnyx Compromised on PyPI: Tracing the TeamPCP Supply Chain Campaign." Datadog Security Labs, 2026.

[19] Snyk. "How a Poisoned Security Scanner Became the Key to Backdooring LiteLLM." Snyk Security Blog, 2026.

[20] InfoQ. "TanStack Details Sophisticated npm Supply Chain Attack That Compromised 42 Packages." InfoQ, May 2026.

[21] Snyk. "Mini Shai-Hulud Hits AntV: 300+ Malicious npm Packages Published via Compromised Maintainer Account." Snyk Security Blog, 2026.

[22] Datadog Security Labs. "Shai-Hulud Goes Open Source." Datadog Security Labs, 2026.

[23] SLSA Project. "Mini Shai-Hulud: Where SLSA's Boundaries Fall." SLSA Blog, May 2026.

[24] Mondoo. "npm Supply Chain Security in 2026: What Your Package Manager Does (and Doesn't) Protect You From." Mondoo Blog, 2026.

[25] Ramimac. "Incident Timeline // TeamPCP Supply Chain Campaign." ramimac.me, 2026.

[26] ChatForest. "TeamPCP Supply Chain Attack: The npm Worm That Hit GitHub, OpenAI, and Mistral AI." ChatForest, 2026.

[27] GitGuardian. "No Off Season: Three Supply Chain Campaigns Hit npm, PyPI, and Docker Hub in 48 Hours." GitGuardian Blog, 2026.

[28] CISA. "2025 Minimum Elements for a Software Bill of Materials (SBOM)." CISA, 2025.

[29] CISA. "Software Bill of Materials for AI — Minimum Elements." CISA, 2025.

[30] Cloud Security Alliance Labs. "Miasma: Red Hat npm Supply Chain Worm." CSA Labs, June 3, 2026.

[31] Cloud Security Alliance Labs. "TeamPCP: Cascading Supply Chain Attack on AI/ML Tooling." CSA Labs, March 30, 2026.

[32] Help Net Security. "LiteLLM PyPI Packages Compromised in Expanding TeamPCP Supply Chain Attacks." Help Net Security, March 25, 2026.

[33] Rescana. "TanStack npm Supply Chain Attack: Detailed Analysis of the May 2026 GitHub Actions Breach and Multi-Ecosystem Impact." Rescana, 2026.

[34] Trend Micro. "Your AI Gateway Was a Backdoor: Inside the LiteLLM Supply Chain Compromise." Trend Micro Research, 2026.

[35] Akamai. "Mini Shai-Hulud: The Worm Returns and Goes Public." Akamai Security Research, 2026.

[36] ReversingLabs. "Team PCP's Mini Shai-Hulud Tears at Open-Source Trust." ReversingLabs Blog, 2026.

[37] SC Media. "NPM v12 to Block Supply-Chain Attacks with New Security Measures." SC World, 2026.

[38] Heise Online. "npm Tackles Its Riskiest Security Issues." Heise Online, 2026.

[39] Phoenix Security. "TeamPCP / Mini Shai-Hulud npm Campaign: 600 Packages, Confirmed Active Payload, Memory Scraping, and 2,500+ Compromised GitHub Repositories." Phoenix Security, 2026.

[40] ThreatLocker. "TeamPCP Supply Chain Attack Hits TanStack." ThreatLocker Blog, 2026.