INTELLIGENCE BRIEFING
Security Command Center
TLP:CLEAR
2026-03-29 18:34 UTC

AI Coding Assistants Introduce Supply Chain Risk via Hallucinated Dependency Recommendations

SECURITY ANALYSIS | MEDIUM | CVSS 5.0

SCC Item ID	SCC-STY-2026-0028
Type	Security Analysis
Severity	MEDIUM
CVSS Base Score	5.0
Affected Products	Software development pipelines using AI coding assistants for dependency management, version recommendations, or patch guidance, no specific product or version; broad applicability across ecosystems (npm, PyPI, Maven, Go, etc.)
Published	2026-03-26
Discovery Source	Rss

Executive Summary

AI coding assistants are generating software dependency recommendations that include fabricated package names, packages that do not exist but can be registered by attackers to deliver malicious code. This supply chain risk operates silently inside development workflows, where AI-generated guidance may pass directly into CI/CD pipelines and container images before any human reviews the dependency list. For security leadership, the signal is structural: AI adoption in the SDLC has outpaced the controls designed to verify what AI recommends.

Technical Analysis

The attack surface here is not a discrete vulnerability, it is a systemic trust failure embedded in how development teams consume AI-generated guidance. When a developer asks an AI coding assistant to recommend a dependency, suggest a version upgrade, or identify a patch, the model may produce a package name that sounds plausible but does not exist in any registry. This phenomenon, sometimes called 'package hallucination,' creates an exploitable gap: an attacker who monitors AI output patterns, or simply registers common-sounding package names speculatively, can position malicious packages to intercept developers who blindly install what the AI recommends.

A Trax Tech analysis, citing broader industry research, estimated that approximately 20% of AI-generated dependency references point to non-existent packages. That figure should be treated as directional rather than

definitive, the underlying research methodology has not been independently validated at the time of this writing, but even a fraction of that rate represents meaningful exposure across large development organizations producing hundreds of dependency decisions per sprint.

The secondary risk layer is less dramatic but arguably more pervasive: AI models recommending outdated or unpatched component versions without flagging known vulnerabilities. A model trained on data predating a CVE disclosure has no mechanism to warn the developer that the version it confidently recommends is insecure. Dark Reading reported on this pattern in March 2026, framing it as accumulated technical debt introduced at the recommendation layer rather than the code layer.

The MITRE ATT&CK mapping reinforces the supply chain framing. T1195.001 (Supply Chain Compromise: Compromise Software Dependencies and Development Tools) is the primary technique; T1072 (Software Deployment Tools) captures the CI/CD propagation vector; T1554 (Compromise Client Software Binary) is relevant where malicious packages modify build artifacts. CWE-1104 (Use of Unmaintained Third-Party Components) and CWE-1357 (Reliance on Insufficiently Trustworthy Component) describe the underlying weakness class.

What makes this risk compound across the SDLC is speed. AI-assisted development is adopted precisely because it accelerates decisions. Dependency recommendations that would previously have required a developer to consult documentation, check vulnerability databases, and verify package provenance now arrive in seconds, and the cognitive overhead of verifying each recommendation erodes with familiarity. Security teams that have not explicitly inserted verification controls between AI recommendation and package installation are operating with an unacknowledged trust boundary inside their pipelines.

No threat actor group has been publicly attributed to a confirmed campaign exploiting this vector at scale. The risk is currently classified as systemic and opportunistic rather than targeted, but the absence of confirmed exploitation does not mean absence of exploitation.

Action Checklist

1. Step 1: Assess exposure, audit which development teams and pipelines use AI coding assistants (GitHub Copilot, Cursor, Claude, ChatGPT, Gemini, or similar) for dependency recommendations, version guidance, or patch suggestions; document the ecosystems in scope (npm, PyPI, Maven, Go modules, etc.)
2. Step 2: Review controls, verify that dependency lockfiles are in use and committed to version control; confirm that software composition analysis (SCA) tools (e.g., Dependabot, Snyk, OWASP Dependency-Check) run automatically on all pull requests and flag packages against known vulnerability databases before merge
3. Step 3: Validate package provenance, require developers to verify that any AI-recommended package actually exists in the target registry before installation; consider enforcing package allowlists or namespace controls in private registries to reduce dependency confusion exposure
4. Step 4: Update threat model, add AI-hallucinated dependency recommendations as an explicit supply chain risk entry in your threat register; map to T1195.001 and T1072; assess whether existing SCA and CI/CD pipeline controls address this vector or were designed only for human-introduced dependency errors
5. Step 5: Communicate findings, brief engineering leadership and AppSec teams on this risk class with specific reference to the ecosystems your organization uses; frame the issue as a process gap (AI

recommendation is not equivalent to verified recommendation) rather than an AI product defect

6. Step 6: Monitor developments, track research publications, registry security reports, and incident disclosures related to package hallucination and dependency confusion; follow CISA supply chain security guidance and MITRE ATT&CK updates for T1195 sub-techniques

IR / Forensic Enrichment

Triage Priority	STANDARD
Escalation Criteria	Escalate to urgent and notify engineering leadership immediately if SCA tooling, CI/CD logs, or developer reports confirm that a package recommended by an AI coding assistant was successfully installed, built into an artifact, or deployed to any environment — even if no malicious payload has yet been confirmed — because the window between attacker package registration and payload delivery can be hours, and any deployed artifact must be treated as potentially compromised pending forensic verification.
Recovery Notes	For any pipeline or environment where a suspected hallucinated package was installed, rebuild all artifacts from source using a clean dependency resolution against a verified registry allowlist, and re-deploy; do not patch in place. Audit all container images and build artifacts produced in the 30 days prior to detection using 'docker history ' and SBOM generation tools (Syft, free, Anchore) to identify whether the suspect package appears in the image layer history. Monitor outbound network connections from CI/CD runners and deployed services for 30 days post-recovery, specifically watching for connections to C2 infrastructure patterns associated with npm/PyPI malware campaigns (random subdomain DNS, high-entropy domain names, connections to pastebin or raw GitHub content URLs) as indicators of a compromised package that executed during build.
Forensic Artifacts	CI/CD pipeline execution logs (GitHub Actions run logs, Jenkins build console output, GitLab CI job traces) showing 'npm install', 'pip install', 'go mod download', or 'mvn dependency:resolve' commands — specifically entries where a package resolved successfully but the package did not exist in the registry at the time it was first recommended by the AI assistant; compare resolution timestamps against registry publish dates via 'npm view time.created' or PyPI JSON API at 'https://pypi.org/pypi//json' | Dependency lockfiles (package-lock.json, Pipfile.lock, go.sum, pom.xml resolved dependency tree) committed to version control in the window when AI assistant usage was active — diff these against registry existence checks to identify any package that has a resolution record but no legitimate registry history predating the commit | Private registry (Artifactory, Nexus, AWS CodeArtifact) access logs or public registry DNS/HTTP proxy logs showing package fetch attempts for names that returned 404 (hallucinated, not yet registered) or 200 (potentially attacker-registered); a package name that went from 404 to 200 between two CI runs is a high-confidence indicator of attacker registration | SBOM (Software Bill of Materials) outputs generated by Syft, CycloneDX CLI, or 'npm sbom --sbom-format cyclonedx' for all container images and build artifacts produced during the exposure window — these provide a timestamped inventory of every package embedded in deployed artifacts that can be cross-referenced against registry malware reports | Developer IDE and AI assistant interaction logs where available — GitHub Copilot telemetry (if enterprise logging is enabled via GitHub Enterprise audit log API), Cursor conversation exports, or browser history for ChatGPT/Gemini sessions — to reconstruct which specific package names were recommended by AI assistants and correlate them against the dependency changes that followed in subsequent commits

Per-Action IR Details

Step 1: Assess exposure — audit which development teams and pipelines use AI coding assistants (GitHub Copilot, Cursor, Claude, ChatGPT, Gemini, or similar) for dependency recommendations, version guidance, or patch suggestions; document the ecosystems in scope (npm, PyPI, Maven, Go modules, etc.)

NIST Phase: Preparation

Reference: NIST 800-61r3 §2 — Preparation: establishing IR capability through asset and process inventory before an incident occurs

Controls: NIST IR-4 (Incident Handling) — implement handling capability that includes preparation for novel supply chain vectors, NIST IR-8 (Incident Response Plan) — update IR plan to enumerate AI-assisted SDLC tooling as an asset class requiring coverage, NIST SA-12 (Supply Chain Protection) — identify and document supply chain elements; AI coding assistants feeding dependency lists are an in-scope supply chain node, CIS 1.1 (Establish and Maintain Detailed Enterprise Asset Inventory) — extend asset inventory to include AI coding assistant integrations (Copilot plugins, Cursor IDE configs, API-connected ChatGPT/Gemini sessions) per team and repo, CIS 2.1 (Establish and Maintain a Software Inventory) — capture which package ecosystems (npm, PyPI, Maven, Go modules) each team operates in, as hallucination risk and attacker registration patterns differ by registry

Compensating: Run 'git log --all --oneline -- package.json package-lock.json requirements.txt pom.xml go.mod' across all active repos to identify recent dependency changes, then cross-reference commit authors with known AI-assistant users. For npm: 'npm ls --all 2>&1 | grep -v deduped' surfaces the full resolved tree. For PyPI: 'pip list --format=freeze > baseline.txt' per project. Maintain a shared spreadsheet mapping team → AI tool → ecosystem; a 2-person team can complete this in one sprint using GitHub org-level repo metadata via 'gh repo list --json name,languages'.

Evidence: Before auditing, snapshot the current dependency state across all repos to establish a pre-audit baseline: capture 'package-lock.json', 'requirements.txt'/'Pipfile.lock', 'go.sum', and 'pom.xml' hashes via 'sha256sum' or 'Get-FileHash' (PowerShell) for every active project. Preserve CI/CD pipeline definitions (GitHub Actions .yml, Jenkinsfile, .gitlab-ci.yml) that show whether dependency installation steps include integrity verification flags such as 'npm ci' vs. 'npm install' or 'pip install --require-hashes'. These baselines are your forensic anchor if a hallucinated package is later found to have been installed.

Step 2: Review controls — verify that dependency lockfiles are in use and committed to version control; confirm that software composition analysis (SCA) tools (e.g., Dependabot, Snyk, OWASP Dependency-Check) run automatically on all pull requests and flag packages against known vulnerability databases before merge

NIST Phase: Preparation

Reference: NIST 800-61r3 §2 — Preparation: verifying that detection and prevention capabilities are operational before exploitation occurs

Controls: NIST SI-2 (Flaw Remediation) — extend flaw remediation process to include SCA gating on PRs; hallucinated packages that resolve to attacker-registered malicious versions are a class of introduced flaw, NIST SA-15 (Development Process, Standards, and Tools) — verify that development tooling standards require SCA integration; AI coding assistant usage without SCA gating violates this control intent, NIST CM-3 (Configuration Change Control) — dependency changes introduced via AI recommendation must pass through the same change control gate as any other configuration change, CIS 7.1 (Establish and Maintain a Vulnerability Management Process) — confirm SCA tools are part of the documented vuln management process, not optional add-ons, CIS 7.4 (Perform Automated Application Patch Management) — validate that automated dependency update tools (Dependabot, Renovate) are not themselves accepting AI-generated version pins without verification

Compensating: For teams without Snyk or commercial SCA: configure OWASP Dependency-Check (free, CLI) as a required step in CI/CD — 'dependency-check.sh --project myapp --scan . --format JSON --out ./reports'. For npm: enforce 'npm audit --audit-level=moderate' as a blocking CI step. For Python: use 'pip-audit' (free, PyPA-maintained) with 'pip-audit -r requirements.txt --output json'. Add a pre-commit hook using 'pre-commit' framework with a custom hook that runs 'pip-audit' or 'npm audit' locally before any dependency change is committed. Lock npm installs to 'npm ci' (requires lockfile) in all pipeline definitions to prevent silent resolution of non-existent packages into attacker-controlled versions.

Evidence: Pull CI/CD pipeline execution logs for the past 90 days from GitHub Actions (Settings → Actions → Workflow runs), GitLab CI job logs, or Jenkins build history — specifically filter for jobs that ran 'npm install', 'pip install', 'mvn dependency:resolve', or 'go mod download' without a preceding SCA scan step. Capture Dependabot or Snyk PR comment history to identify packages flagged but merged anyway. In GitHub: 'gh api repos/{owner}/{repo}/pulls?state=merged' with filter on label 'dependencies' to surface unreviewed dependency PRs.

Step 3: Validate package provenance — require developers to verify that any AI-recommended package actually exists in the target registry before installation; consider enforcing package allowlists or namespace controls in private registries to reduce dependency confusion exposure

NIST Phase: Containment

Reference: NIST 800-61r3 §3.3 — Containment Strategy: implementing controls that limit the spread or impact of a threat while longer-term solutions are developed

Controls: NIST SI-7 (Software, Firmware, and Information Integrity) — enforce integrity verification for all packages at install time; for npm use '--ignore-scripts' plus lockfile hash verification; for PyPI enforce '--require-hashes' in requirements files, NIST CM-7 (Least Functionality) — restrict package installation to allowlisted namespaces in Artifactory, Nexus, or AWS CodeArtifact, denying resolution against public registries for packages not on the approved list, NIST SA-12 (Supply Chain Protection) — namespace controls and private registry mirroring are explicit supply chain protection mechanisms against dependency confusion, the attacker technique enabled by hallucinated package names, CIS 2.3 (Address Unauthorized Software) — any package not present in the approved software inventory (including registry allowlist) should be blocked at install time, not flagged post-merge, CIS 4.6 (Securely Manage Enterprise Assets and Software) — manage dependency resolution configuration (npmrc, pip.conf, settings.xml, GONOSUMCHECK) as a secured configuration artifact under version control

Compensating: For npm: add '.npmrc' to each repo root with 'registry=https://your-private-registry' and set 'npm config set package-lock true'; use 'npm pack ' to dry-run resolution before committing — a 404 from the registry confirms the package does not exist. For PyPI: configure 'pip.conf' with '--index-url' pointing to a mirrored registry (devpi is free and self-hosted). For Go: set 'GONOSUMCHECK' and 'GOFLAGS=-mod=readonly' in CI to prevent resolution of unlocked modules. Publish a one-page developer checklist: before adding any AI-recommended dependency, manually visit 'https://www.npmjs.com/package/' or 'https://pypi.org/project/' and verify the package owner, publish date, and download count are consistent with a legitimate, maintained library.

Evidence: Capture network proxy or DNS query logs from developer workstations and CI runners for the 30 days prior to this control implementation — specifically DNS lookups or HTTP requests to 'registry.npmjs.org', 'pypi.org', 'repo1.maven.org', or 'proxy.golang.org' for package names that return NXDOMAIN or HTTP 404. These failed resolutions are the forensic signature of hallucinated package installation attempts; a successful resolution for a package with a creation date newer than the AI recommendation date is the signature of an attacker who pre-registered the hallucinated name. Also preserve any Artifactory or Nexus access logs showing package pull attempts for packages not in the approved catalog.

Step 4: Update threat model — add AI-hallucinated dependency recommendations as an explicit supply chain risk entry in your threat register; map to T1195.001 and T1072; assess whether existing SCA and CI/CD pipeline controls address this vector or were designed only for human-introduced dependency errors

NIST Phase: Preparation

Reference: NIST 800-61r3 §2 — Preparation: maintaining current threat intelligence and risk documentation to ensure IR capability addresses active threat vectors

Controls: NIST RA-3 (Risk Assessment) — formally document AI hallucination as a supply chain threat source in the organizational risk assessment; assign likelihood and impact specific to the ecosystems in use, NIST IR-8 (Incident Response Plan) — update IR plan with a specific scenario: 'malicious package installed via AI-hallucinated dependency recommendation' with detection indicators and response procedures distinct from traditional dependency confusion playbooks, NIST PM-16 (Threat Awareness Program) — incorporate MITRE ATT&CK T1195.001 (Compromise Software Dependencies and Development Tools) and T1072 (Software Deployment Tools) into the threat awareness program with AI-specific attack path documentation, CIS 7.1 (Establish and Maintain a Vulnerability Management Process) — update the vuln management process to treat AI coding assistant outputs as an untrusted input source requiring the same verification rigor as third-party code

Compensating: Use the free MITRE ATT&CK Navigator (https://mitre-attack.github.io/attack-navigator/) to annotate T1195.001 and T1072 with organization-specific notes on AI-assisted attack paths. Document the threat register entry in a shared wiki or markdown file in a security repo using this structure: Threat ID, Attack Path (AI recommends non-existent package → developer installs → attacker-registered malicious version executes in CI → build artifact compromised), Affected Ecosystems, Current Controls, Control Gap (SCA checks known-bad packages but cannot detect newly registered attacker packages with zero vulnerability history), Residual Risk. A 2-person team can complete this in a single threat modeling session using the STRIDE or PASTA methodology applied specifically to the CI/CD pipeline dataflow.

Evidence: Before updating the threat model, pull the current threat register and SCA tool configuration to document the pre-existing control gap as evidence: export current Dependabot or OWASP Dependency-Check rule sets and confirm they operate by comparing package names against known vulnerability databases (NVD, OSV) — this documents that they would not flag a newly registered, zero-history malicious package. This gap documentation is your evidence baseline for the threat model update and provides audit trail for why the control gap exists.

Step 5: Communicate findings — brief engineering leadership and AppSec teams on this risk class with specific reference to the ecosystems your organization uses; frame the issue as a process gap (AI recommendation is not equivalent to verified recommendation) rather than an AI product defect

NIST Phase: Post Incident

Reference: NIST 800-61r3 §4 — Post-Incident Activity: lessons learned and communication to improve organizational security posture and prevent recurrence

Controls: NIST IR-6 (Incident Reporting) — establish reporting channels for developers to flag suspected AI-hallucinated package recommendations without friction; normalize reporting before an incident occurs, NIST IR-2 (Incident Response Training) — include AI hallucination supply chain scenario in developer security training; engineers using GitHub Copilot, Cursor, or ChatGPT for dependency guidance must understand that AI output is a suggestion, not a verified package reference, NIST AT-2 (Literacy Training and Awareness) — update security awareness content to address AI coding assistant risks specific to the npm, PyPI, and other ecosystems in use; include concrete examples of hallucinated package name patterns, CIS 7.2 (Establish and Maintain a Remediation Process) — communicate the remediation process (verify before install, SCA gate, registry allowlist) to all developers as a documented, required workflow change, not an optional best practice

Compensating: Prepare a one-page internal advisory with two concrete examples: (1) a hallucinated npm package name pattern (e.g., a plausible-sounding scoped package like '@company-utils/auth-helper' that does not exist in the registry) and (2) a real dependency confusion incident (the Alex Birsan 2021 research is publicly documented and ecosystem-agnostic) to make the attack vector tangible. Deliver via existing engineering all-hands or Slack/Teams channel; do not require a separate security meeting. Include a mandatory action item: 'Before merging any PR that adds a new dependency recommended by an AI assistant, paste the package name into [registry URL] and verify it exists with a publish history older than 6 months.'

Evidence: Before the communication, document which teams received the briefing and when (attendance log, Slack message timestamp, email send receipt) to satisfy NIST IR-2 training documentation requirements. Capture the pre-briefing state of any developer surveys or code review checklists to establish a baseline for measuring whether behavior changes post-communication. This record supports post-incident audit requirements under NIST AU-11 (Audit Record Retention) if a hallucination-enabled incident later occurs.

Step 6: Monitor developments — track research publications, registry security reports, and incident disclosures related to package hallucination and dependency confusion; follow CISA supply chain security guidance and MITRE ATT&CK updates for T1195 sub-techniques

NIST Phase: Post Incident

Reference: NIST 800-61r3 §4 — Post-Incident Activity: using threat intelligence to improve detection capability and update the IR program continuously

Controls: NIST SI-5 (Security Alerts, Advisories, and Directives) — establish a formal process to receive and act on CISA supply chain security advisories and registry security reports (npm security advisories, PyPI malware reports) related to dependency confusion and hallucination-enabled packages, NIST IR-5 (Incident Monitoring) — extend

incident monitoring to include registry-level threat feeds; npm publishes security advisories at https://github.com/advisories and PyPI publishes malware removal notices that serve as early warning for attacker-registered packages, NIST PM-16 (Threat Awareness Program) — subscribe to MITRE ATT&CK update notifications for T1195 (Supply Chain Compromise) sub-technique additions that may formalize AI hallucination as a distinct attack path, CIS 7.1 (Establish and Maintain a Vulnerability Management Process) — include registry security feeds (npm, PyPI, OSV.dev, deps.dev) as formal intelligence sources in the vulnerability management process, reviewed on a cadence no less frequent than monthly

Compensating: Set up free RSS or GitHub watch notifications on: (1) github.com/pypa/advisory-database for PyPI malware reports, (2) github.com/nicowillis/security-advisories or the npm security feed at https://github.com/advisories?query=type%3Amalware for npm, (3) OSV.dev for cross-ecosystem vulnerability and malware disclosures. Subscribe to CISA's free email alerts at cisa.gov/news-events/cybersecurity-advisories. Use a free OSINT aggregator (e.g., Feedly free tier or a self-hosted RSS reader) to consolidate research publications from arXiv (cs.CR category), academic blogs covering AI security, and the deps.dev Open Source Insights project. A 2-person team can assign one person a 15-minute weekly review cadence for these feeds.

Evidence: Maintain a dated intelligence log (markdown file in a security repo is sufficient) recording each relevant advisory, research paper, or incident disclosure reviewed, with a note on whether it introduced a new indicator (new hallucinated package name pattern, new attacker-registered namespace, new ecosystem affected) that requires updating detection rules or SCA configurations. This log serves as evidence of continuous monitoring activity under NIST IR-5 (Incident Monitoring) and provides an audit trail demonstrating that the organization actively tracked this emerging threat class.

Detection Guidance

There are no traditional IOCs for this risk class, no malicious IP, no known hash. Detection must focus on process anomalies and dependency integrity signals.

In your package registries and CI/CD pipelines: alert on first-time dependencies introduced without a corresponding code review comment or documented rationale; flag packages with zero download history, no public repository, no maintainer history, or registration dates within days of first use in your codebase. These patterns suggest a newly registered package that may have been placed speculatively to intercept a hallucinated recommendation.

In your SCA tooling: review reports for packages that resolve successfully in the registry but have no vulnerability history, no release history older than 90 days, and no community activity, this profile matches a placeholder package registered to capture hallucinated names.

In your build and CI/CD logs: hunt for dependency resolution events that were not present in a prior lockfile state and were not accompanied by a human-authored commit message explaining the addition. AI-generated code commits may lack this context.

For version integrity: cross-reference AI-recommended versions against the National Vulnerability Database (NVD) and the relevant ecosystem's security advisories (npm audit, pip-audit, OSV.dev) before any installation. Flag cases where an AI recommendation matches a version with a known CVE.

Policy gap to audit: determine whether your developer guidelines explicitly address AI-generated dependency recommendations and whether SCA tooling is positioned before, not after, package installation in your pipeline sequence.

Framework Mappings

MITRE-ATTACK

- **T1526** — Cloud Service Discovery
- **T1072** — Software Deployment Tools
- **T1195.001** — Compromise Software Dependencies and Development Tools
- **T1059** — Command and Scripting Interpreter
- **T1554** — Compromise Host Software Binary

NIST-800-53R5

- **CM-7** — Least Functionality
- **SI-3** — Malicious Code Protection
- **SI-4** — System Monitoring
- **SI-7** — Software, Firmware, and Information Integrity
- **SA-4** — Acquisition Process
- **SA-9** — External System Services
- **AT-2** — Literacy Training and Awareness
- **SR-2** — Supply Chain Risk Management Plan

OWASP-TOP10-2021

- **A06:2021** — Vulnerable and Outdated Components

CIS-V8

- **16.4**
- **7.3** — Perform Automated Operating System Patch Management
- **7.4** — Perform Automated Application Patch Management
- **14.2** — Train Workforce Members to Recognize Social Engineering Attacks
- **15.1** — Establish and Maintain an Inventory of Service Providers
- **8.2** — Collect Audit Logs

ISO-27001-2022

- **A.8.8** — Management of technical vulnerabilities
- **A.5.21** — Managing information security in the ICT supply chain

NIST-CSF-2

- **GV.SC-01** — Cybersecurity supply chain risk management program
- **DE.CM-01** — Networks and network services are monitored

SOC2-TSC

- **CC9.2** — Manages risks associated with vendors and business partners

MITRE ATT&CK Mapping

Technique ID	Technique Name	Tactic
T1526	Cloud Service Discovery	Discovery

Technique ID	Technique Name	Tactic
T1072	Software Deployment Tools	Execution
T1195.001	Compromise Software Dependencies and Development Tools	Initial-Access
T1059	Command and Scripting Interpreter	Execution
T1554	Compromise Host Software Binary	Persistence

Sources

Source	URL	Tier
Security News	https://www.darkreading.com/application-security/ai-powered-depende...	T3
AI-Generated Code Security Risks: What Developers Must Know	https://www.veracode.com/blog/ai-generated-code-security-risks/	T3
Implications of Code-Gen AI tools for developers : r/cybersecurity	https://www.reddit.com/r/cybersecurity/comments/1jfa39r/implication...	T3
AI coding tools exploded in 2025. The first security exploits show ...	https://fortune.com/2025/12/15/ai-coding-tools-security-exploit-sof...	T3
20% of AI-Generated Code Dependencies Don't Exist, Creating ...	https://www.traxtech.com/blog/20-of-ai-generated-code-dependencies-...	T3

Generated 2026-03-29 18:34 UTC by TJS Security Command Center