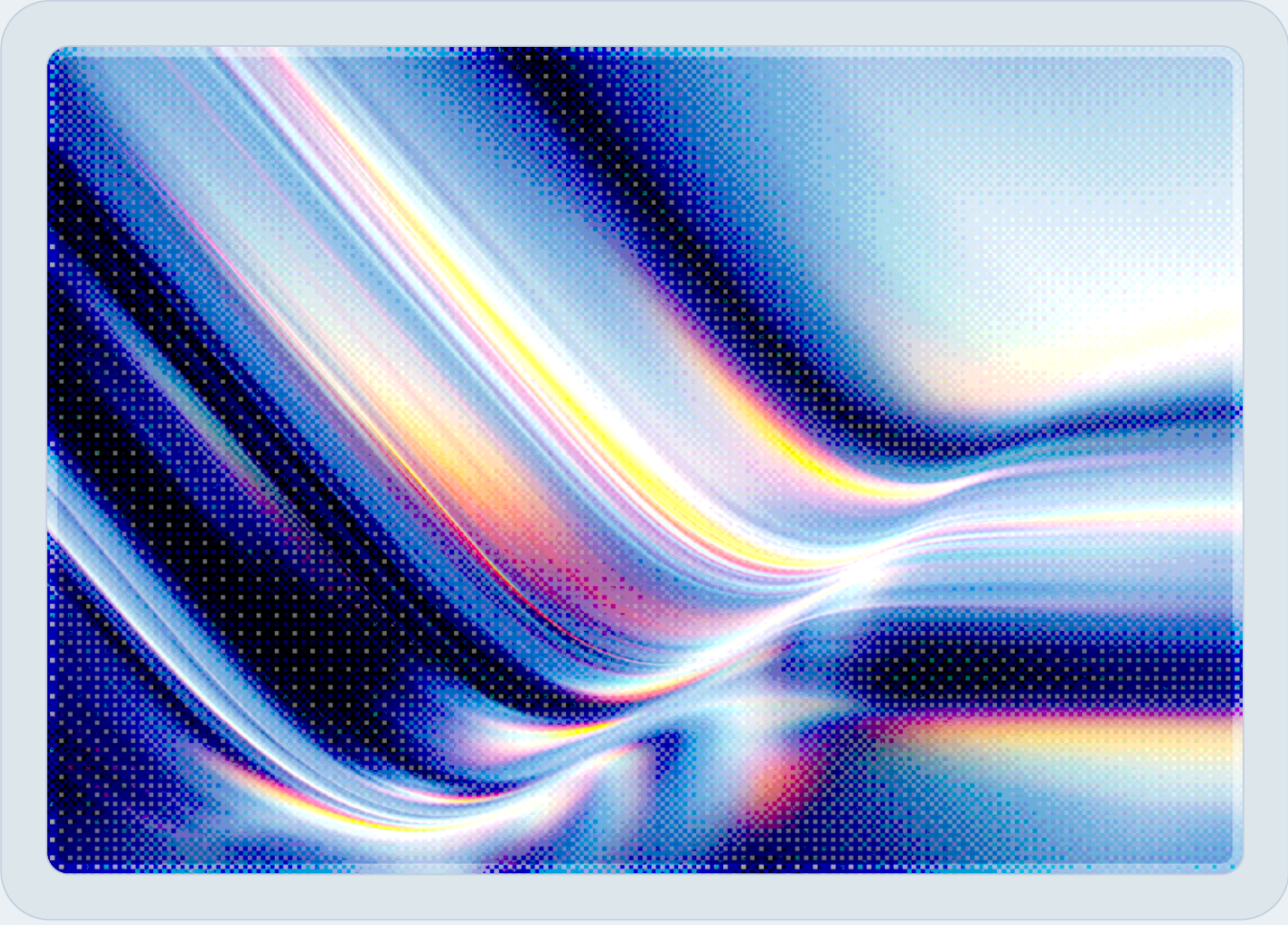
CSAI Foundation | Cloud Security Alliance

Pinned but Not Frozen: Dependency Drift in Coding-Assistant MCP Servers

Security Implications and Guidance for Package-Launched MCP Servers

2026-11-15

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

© 2026 Cloud Security Alliance. Some rights reserved.

You may download, store, display, view, print, redistribute, and link to this document in its original, unmodified form, provided that attribution to the Cloud Security Alliance is maintained and all trademark and copyright notices remain intact.

This document may not be modified or altered. You may quote portions of the document as permitted by the Fair Use provisions of the United States Copyright Act, provided that attribution is given to the Cloud Security Alliance.

This document may be shared on professional and social media platforms in its original form with attribution.

This document was generated with AI assistance and has not undergone official CSA review and approval processes.

Key Takeaways

Many local Model Context Protocol (MCP) servers used by AI coding assistants are not installed in the usual sense. They are launched. A configuration entry such as `"command": "npx", "args": ["-y", "@example/mcp-server"]` tells the assistant to fetch a package from a public registry, resolve its dependency tree, and run it as a child process with the developer's privileges. Claude Code's documentation uses this pattern in its stdio examples [1]. In Corelight's review of 35 MCP servers, 23 had a recommended install that would resolve dependencies from a public registry, either on every launch or on each install or image build [26]. As a result, the code behind an MCP server can change between sessions without any edit to the configuration file the security team reviewed. This note calls that gap *dependency drift*.

Pinning the top-level version narrows the gap but does not close it. npm never publishes a package's `package-lock.json` [5], so `npx -y some-server@1.4.2` fixes only the server's own code. Its transitive dependencies are still resolved fresh against semver ranges whenever a new environment is built. Python servers launched with `uvx` work the same way: an exact pin on the tool does not lock the tool's dependencies, and the cached environment is disposable [3][4]. Corelight observed this directly. A server configured to run one release, built from one commit, ran with 15 different dependency sets over five weeks [26]. A pinned MCP server is therefore pinned but not frozen. The March 2026 axios compromise, which pushed a remote access trojan through an injected transitive dependency during a window of roughly three hours [13][14], shows how a single upstream release can reach any fresh resolution made during that window whose version range admits it.

The recommended response has three parts. First, pin MCP servers exactly and inventory them as executable dependencies. Second, add release-age cooldowns and disable install scripts in the environments where assistants launch servers. Third, move high-value servers to artifacts that are immutable all the way down, such as locked internal builds or container images referenced by digest. Teams should also remember that pinning freezes vulnerable versions as well as safe ones, as CVE-2025-6514 in `mcp-remote` showed [8][9]. Pinning without a tracked update process trades one risk for another.

Background

MCP lets an AI application connect to external tools and data through "servers." Remote servers run over HTTP. Local servers run as subprocesses that communicate over standard input and output (stdio). AI coding assistants, including Claude Code, Cursor, GitHub Copilot in VS Code, and others, read server definitions from user-level or project-level configuration files. Claude Code, for example, supports a project-scoped `.mcp.json` file that is committed to the repository and shared with everyone who clones it [1]. Its documentation's stdio examples launch community servers with `npx -y`. The `-y` flag suppresses npx's normal confirmation prompt before it installs a package that is not present locally [1][2].

Package runners were made for convenience, and their resolution behavior shows it. When npx runs a package that is not among the local project's dependencies, it installs that package into a folder in the npm cache and runs it from there [2]. A bare package name resolves against the registry's `latest` tag. Whether a particular launch fetches a newer release depends on local cache state and npm version, which is exactly the kind of implicit behavior that makes fleet-wide reasoning hard. uv's `uvx` is more explicit. On first invocation it uses the latest version, then reuses a cached environment until the cache is pruned or refreshed or the user requests `@latest` [3]. The uv documentation describes that cached environment as disposable: `uv cache clean` deletes it, and a new one is created automatically on the next run [3].

Locking matters here. npm's documentation states that `package-lock.json` is never included in a published package, even if listed in the `files` field. The publishable alternative, `npm-shrinkwrap.json`, is discouraged for libraries and recommended mainly for applications and command-line tools [5]. MCP servers are effectively CLI tools, but nothing requires their publishers to ship a shrinkwrap, and a server published without one leaves its transitive tree to be resolved at install time. Python has a similar gap: published wheels declare dependency ranges in their metadata, not a lock. uvx accepts exact pins through `tool@version` or `--from 'tool==version'` [4], but those specifiers constrain only the named package.

The official MCP Registry partly addresses the top layer. Its versioning rules require every published server version to be unique and immutable once published, prohibit range specifiers such as `^1.2.3`, and leave it to the registry to decide which version is marked "latest" [19]. Those rules govern registry metadata. They do not govern the dependency tree that a package manager resolves when the assistant launches the server.

Security Analysis

The mechanics described above produce a specific and somewhat counterintuitive exposure profile. This section first separates the layers at which drift can enter a package-launched server, then reviews what measurement and the recent incident record show about each layer. It closes by examining why drift is not a one-directional risk and why the coding-assistant context makes it more consequential than it would be for an ordinary developer tool.

Three Layers of Drift

Drift in a package-launched MCP server can enter at three layers, and each needs a different control. The table below summarizes how each layer behaves under common launch patterns.

Layer	What can change without a config edit	`npx -y pkg` (unpinned)	`npx -y pkg@1.4.2` / `uvx pkg@1.4.2` (pinned)	Locked build or image by digest
Server code	New release of the server package itself	Exposed whenever a fresh resolution occurs	Fixed	Fixed
Transitive dependencies	New releases within declared semver ranges	Exposed	Exposed whenever the environment is rebuilt (cache miss, cache clean, new machine, CI runner)	Fixed
Runtime and launcher	Node/Python interpreter, npm/uv version, registry or mirror configuration	Exposed	Exposed	Largely fixed (image) or partially fixed (locked build on host runtime)

The middle column is the one this note is concerned with. Version pinning can create the impression that a server is fixed. In practice, the transitive layer of an MCP server can be large: in the commit-pinned server Corelight tracked, 23 of its 69 packages changed versions over five weeks, including the cryptography library, the JSON Web Token library (four versions), and the MCP SDK itself [26]. Each of those packages is resolved against a caret or tilde range, so the pinned server's behavior can change whenever any of them ships a new release that a fresh environment picks up. Cache state determines exposure, and rebuilds are frequent. Corelight found 19 builds of the same release and commit in the launcher's cache, made about every couple of days, and its endpoint telemetry recorded 17 to 21 rebuilds per MCP server over 30 days [26]. Because cache state also differs across developer laptops, ephemeral cloud workspaces, and CI runners, exposure is hard to reconstruct after an incident.

What the Incident Record Shows

Several incidents from 2025 and 2026 map onto these layers. The first malicious MCP server reported in the wild, an npm package named `postmark-mcp` that impersonated Postmark's email server, published 15 clean versions before version 1.0.16 added a single line that blind-copied every outgoing email to an attacker-controlled address [6][7]. The package attracted 1,643 total downloads before removal, according to reporting on Koi Security's research [6]. Users who had evaluated an earlier version and then launched the server unpinned could have received the backdoored release on their next fresh resolution, without any visible change in their configuration. This is drift at the server-code layer. It is also the "rug pull" pattern that MCP threat models describe in terms of tool metadata, carried out here through ordinary package versioning.

The transitive layer is shown by the axios compromise of March 31, 2026. An attacker who had taken over a maintainer's npm credentials published axios 1.14.1 and 0.30.4. Neither release changed axios's own source. Instead, each added a dependency on `plain-crypto-js@4.2.1`, a typosquat whose `postinstall` script fetched a cross-platform remote access trojan [13][14]. The malicious versions were reportedly available for roughly three hours before removal [13][20]. Public reporting does not attribute any compromise specifically to an MCP server. However, the semver mechanics suggest that a pinned MCP server declaring a dependency such as `axios@^1.x` would have resolved to the malicious release if its environment had been rebuilt during that window. This is an inference from how ranges resolve, not a documented case, but it is the scenario that exact top-level pinning does not address.

Two further incidents are cited here not as drift cases, since neither involved an MCP server, but because they show why coding-assistant environments are attractive targets for install-time payloads. In the Nx "s1ngularity" compromise of August 26, 2025, malicious package versions ran a post-install script that harvested secrets and invoked locally installed AI command-line tools, including Claude Code, Gemini CLI,

and Amazon Q, with flags that bypass permission prompts, so that the assistants would help search the file system [10][11]. The packages were live for about four hours [10]. A month later, CISA warned of the self-replicating "Shai-Hulud" worm, which had compromised more than 500 npm packages by harvesting credentials and using stolen tokens to publish poisoned versions [12][24]. Short exposure windows, install-time execution, and developer workstations full of credentials are conditions that make unfrozen dependency resolution particularly hazardous.

Drift Cuts Both Ways

A policy of "pin everything and never update" is not a safe end state. CVE-2025-6514 affected `mcp-remote`, a proxy that lets local MCP clients connect to remote servers. In versions 0.0.5 through 0.1.15, a malicious server could return a crafted OAuth authorization endpoint that led to arbitrary operating system command execution on Windows clients and more limited executable launch on macOS and Linux. JFrog rated the flaw CVSS 9.6, and it was fixed in 0.1.16 [8][9]. An organization that had pinned `mcp-remote@0.1.15` across its fleet would have frozen a critical vulnerability in place. The aim is controlled change: updates that are deliberate, reviewed, and reproducible, rather than implicit updates driven by cache misses.

Why Coding Assistants Amplify the Problem

Three features of the coding-assistant context make drift more consequential than it would be for a typical developer CLI. First, launch is automatic and often unattended. Claude Code prompts for approval of project-scoped servers in interactive sessions, but its documentation notes that in `claude -p` runs, Agent SDK sessions, and cloud sessions outside a project's threads it cannot show that prompt and loads project-scoped servers without asking [1]. Launches are also frequent: Corelight reports that the assistant started one tracked server 162 times in a single week [26]. Second, the configuration that defines the launch command travels with the repository. A developer who clones a project inherits its server list, and CSA's prior analysis of MCP launch paths concluded that any field determining what a client spawns should be treated as a code-execution primitive [21]. CSA's earlier note on MCP auto-execution documented how project-scoped server settings launched at startup, before any trust check, led to cloud credential theft from a cloned repository [27].

Third, the assistant adds its own credentials and permissions to what a launched server can reach. A server process inherits the developer's usual environment of cloud credentials, Git tokens, and SSH keys, as any CLI would, but in an assistant session it also sits alongside the assistant's API keys and whatever agent permissions the session has been granted, and it typically runs as a long-lived process for the whole session. Approval mechanisms do not close this gap. Claude Code's project-scope approval [1] attaches

to a server's configuration entry, and none of the client documentation reviewed for this note describes binding approval to a content hash of the code that will run. An approval granted on Monday therefore does not attest to what launches on Thursday.

Recommendations

The recommendations below follow the three-part response outlined in the Key Takeaways and are organized by time horizon. Immediate actions reduce exposure using controls that most package managers already provide. Short-term mitigations replace ad hoc resolution with locked artifacts and observable registry traffic, and strategic considerations address the gap between approval and execution that only assistant vendors and server publishers can close.

Immediate Actions

Organizations should begin by inventorying every MCP server definition across user-level configuration files, project `.mcp.json` and equivalent files in repositories, and any managed or enterprise settings the assistant supports. For each entry, record the launch command, package name, and whether a version is specified. Entries that call `npx`, `uvx`, `pipx run`, `bunx`, or similar runners without an exact version should be treated as unpinned and corrected first, using `package@x.y.z` for npx and `tool@x.y.z` or `--from 'tool==x.y.z'` for uvx [4]. Teams should also check for `mcp-remote` at versions below 0.1.16 and upgrade any they find [8].

Next, apply release-age cooldowns wherever MCP servers are resolved. npm supports `min-release-age`, expressed in days, and `before`, an absolute date [15]. One practitioner guide states that `min-release-age` requires npm 11.10.0 or later [20], and administrators should confirm that requirement against npm's release notes for the versions they deploy. pnpm offers `minimumReleaseAge` in minutes, introduced in version 10.16 [16][17]; its current documentation gives a default of 1,440 minutes (one day) [16]. Bun provides an equivalent setting in seconds [20], and uv's `exclude-newer` accepts friendly durations such as `7 days` [18]. Administrators should verify that their chosen setting is honored by the specific runner code path (`npx` / `npm exec`, `uvx`) in the versions deployed, because documentation for these settings generally describes install and resolution commands rather than ad hoc runners. Where servers do not require install scripts, setting `ignore-scripts=true` closes the lifecycle-script channel that the Nx and axios payloads used [10][13][20], though it does not prevent malicious code in a dependency from running when the server itself imports it.

Short-Term Mitigations

Over the following one to three months, organizations should move frequently used servers off ad hoc resolution. The simplest pattern is an internal "wrapper" package or project for each approved server that carries its own committed lockfile (`package-lock.json`, `uv.lock`). The assistant configuration then points at a locally built entry point instead of `npx`. Publishing an approved server internally with an `npm-shrinkwrap.json` achieves the same result for Node.js tools, which matches npm's own guidance that shrinkwrap is appropriate for CLI tools [5]. Corelight's validation of this approach is encouraging: five frozen servers started 457 times over two weeks ran with installed package lists that matched those recorded on install day [26]. Container images referenced by digest rather than tag provide the most complete immutability of the options here, because they fix the runtime and launcher layer as well as the dependency tree, and they also give a natural place for sandboxing and egress restrictions. Like any pin, however, they still require the tracked update process described below.

Organizations should also route registry traffic for developer and agent environments through an internal proxy or artifact repository. A proxy provides a single point for applying cooldowns, blocking known-malicious versions quickly, and keeping logs of which versions were actually resolved. Those logs are the evidence needed to answer "were we exposed?" when the next three-hour window occurs. MCP configuration files should be classified as security-relevant code: changes that add a server or alter a launch command should require review, and CI should flag unpinned runner invocations. CSA's analysis of the Deadbugz campaign, which distributed MCP configurations through pull requests, supports this kind of review gate [23].

Strategic Considerations

Over the longer term, the gap between approving a server and knowing what code runs should be closed at the client. Assistant vendors could bind approval to a content digest of the resolved server and its dependency tree, re-prompting when that digest changes. This would be analogous to the detection of drift between a tool's approved definition and its runtime behavior that CSA's Deadbugz analysis recommended [23]. MCP server publishers can contribute by shipping a publishable lockfile or a container image with each release, and the MCP Registry's immutable-version rules [19] could reasonably extend to declaring such artifacts. Procurement and platform teams should ask assistant vendors how their products resolve, cache, and verify locally launched servers, and whether enterprise policy can prohibit unpinned launches.

Finally, organizations need an update process that is as deliberate as the pinning. A small owning team should track advisories for approved servers and their dependency trees, test updates on a defined cadence, and roll them out through the same pinned channel. This approach responds to cases like CVE-

2025-6514, where staying current was the security control, without reopening the implicit-update path that cases like `postmark-mcp` exploited.

CSA Resource Alignment

This note builds most directly on CSA's research note *MCP Launch Paths as RCE: The opencode OAuth Flaw and Copilot CVE-2026-103572* [21]. That analysis concluded that any file or field determining what an MCP client spawns should be treated as a code-execution primitive, and that validation of one representation does not secure a different one that is later executed. Dependency drift extends that argument along the time axis. Even when the launch command is reviewed and unchanged, the code it resolves to can change, so the reviewed representation and the executed one diverge. CSA's *Plugin4Shell: SHA-Pinning Bypass Enables AI Coding Agent RCE* [28] reached a closely related conclusion from a different direction: four major AI coding agents locked plugins to a reviewed commit hash but did not check that the code written to disk matched that hash. In both cases, a pin that appears intact does not guarantee that the reviewed code is the code that runs.

CSA's note *15,465 MCP Servers and No Consistent Governance Signal: Marketplace Supply Chain Risk* [22] recommends treating every MCP server connection as a third-party code and data dependency governed through an allowlist. The findings here suggest that an allowlist entry should specify a server name together with an exact version and, where possible, a locked dependency tree or image digest. Otherwise the allowlisted entity is a moving target. Similarly, *Deadbugz: Active MCP Campaign Poisons Agents After Trust* [23] described a server that changed behavior after initial approval and recommended monitoring for drift between a tool's approved definition and its runtime behavior. Package-level drift is the supply chain counterpart of that runtime pattern, and the same principle applies: one-time approval is not continuing assurance. CSA's notes on MCP auto-execution in AI coding assistants [27] and on the Shai-Hulud worm's spread through AI coding assistants [24] provide further context on how repository-carried configuration and registry-scale compromises reach developer agent environments.

At the control-framework level, the recommendations map to four domains of the CSA AI Controls Matrix (AICM) v1.1 [25]: Supply Chain Management, Transparency, and Accountability (STA); Application and Interface Security (AIS); Change Control and Configuration Management (CCC); and Threat and Vulnerability Management (TVM). Inventory, pinning, and provenance requirements correspond to supply chain controls. Cooldowns and script restrictions correspond to application security controls, while review of MCP configuration changes corresponds to change control. The tracked update process corresponds to vulnerability management. Organizations already assessing themselves against AICM can treat package-launched MCP servers as an explicit asset class within those domains rather than as an informal developer convenience.

References

[1] Anthropic. "Connect Claude Code to tools via MCP." Claude Code Documentation, accessed November 2026.

[2] npm, Inc. "npx." npm CLI v11 Documentation, accessed November 2026.

[3] Astral. "Tools." uv Documentation, accessed November 2026.

[4] Astral. "Using tools." uv Documentation, accessed November 2026.

[5] npm, Inc. "npm-shrinkwrap.json." npm CLI v11 Documentation, accessed November 2026.

[6] The Hacker News. "First Malicious MCP Server Found Stealing Emails in Rogue Postmark-MCP Package." The Hacker News, September 2025 (reporting Koi Security research).

[7] The Register. "Fake Postmark MCP npm package stole emails with one-liner." The Register, September 29, 2025.

[8] JFrog Security Research. "Critical RCE Vulnerability in mcp-remote: CVE-2025-6514 Threatens LLM Clients." JFrog Blog, July 2025.

[9] NIST. "CVE-2025-6514 Detail." National Vulnerability Database, 2025.

[10] Nx. "S1ngularity - What Happened, How We Responded, What We Learned." Nx Blog, 2025.

[11] Snyk. "Weaponizing AI Coding Agents for Malware in the Nx Malicious Package." Snyk Blog, August 2025.

[12] CISA. "Widespread Supply Chain Compromise Impacting npm Ecosystem." Cybersecurity and Infrastructure Security Agency, September 23, 2025.

[13] Datadog Security Labs. "Compromised axios npm package delivers cross-platform RAT." Datadog Security Labs, March 2026.

[14] Snyk. "Axios npm Package Compromised: Supply Chain Attack Delivers Cross-Platform RAT." Snyk Blog, March 2026.

[15] npm, Inc. "config." npm CLI v11 Documentation, accessed November 2026.

[16] pnpm. "Mitigating supply chain attacks." pnpm Documentation, accessed November 2026.

[17] Socket. "pnpm 10.16 Adds New Setting for Delayed Dependency Updates." Socket Blog, September 2025.

[18] Astral. "Settings: exclude-newer." uv Documentation, accessed November 2026.

[19] Model Context Protocol. "Versioning Published MCP Servers." MCP Registry Documentation, accessed November 2026.

[20] Reehal, Jag. "The package manager settings that would have blocked the Axios attack." Arrange Act Assert, April 7, 2026.

[21] Cloud Security Alliance. "MCP Launch Paths as RCE: The opencode OAuth Flaw and Copilot CVE-2026-103572." CSA AI Safety Initiative, October 22, 2026.

[22] Cloud Security Alliance. "15,465 MCP Servers and No Consistent Governance Signal: Marketplace Supply Chain Risk." CSA AI Safety Initiative, October 7, 2026.

[23] Cloud Security Alliance. "Deadbugz: Active MCP Campaign Poisons Agents After Trust." CSA AI Safety Initiative, August 30, 2026.

[24] Cloud Security Alliance. "AI Coding Assistant Hijack Spreads Shai-Hulud Worm." CSA AI Safety Initiative, September 18, 2026.

[25] Cloud Security Alliance. "AI Controls Matrix (AICM) v1.1." Cloud Security Alliance, 2026.

[26] Lagermann, James. "The supply-chain risk hiding in your MCP servers." Corelight Blog, November 10, 2026.

[27] Cloud Security Alliance. "MCP Auto-Execution: AI Coding Assistants and Credential Theft." CSA AI Safety Initiative, June 29, 2026.

[28] Cloud Security Alliance. "Plugin4Shell: SHA-Pinning Bypass Enables AI Coding Agent RCE." CSA AI Safety Initiative, September 19, 2026.

© 2026 Cloud Security Alliance. AI-assisted Rapid Research.