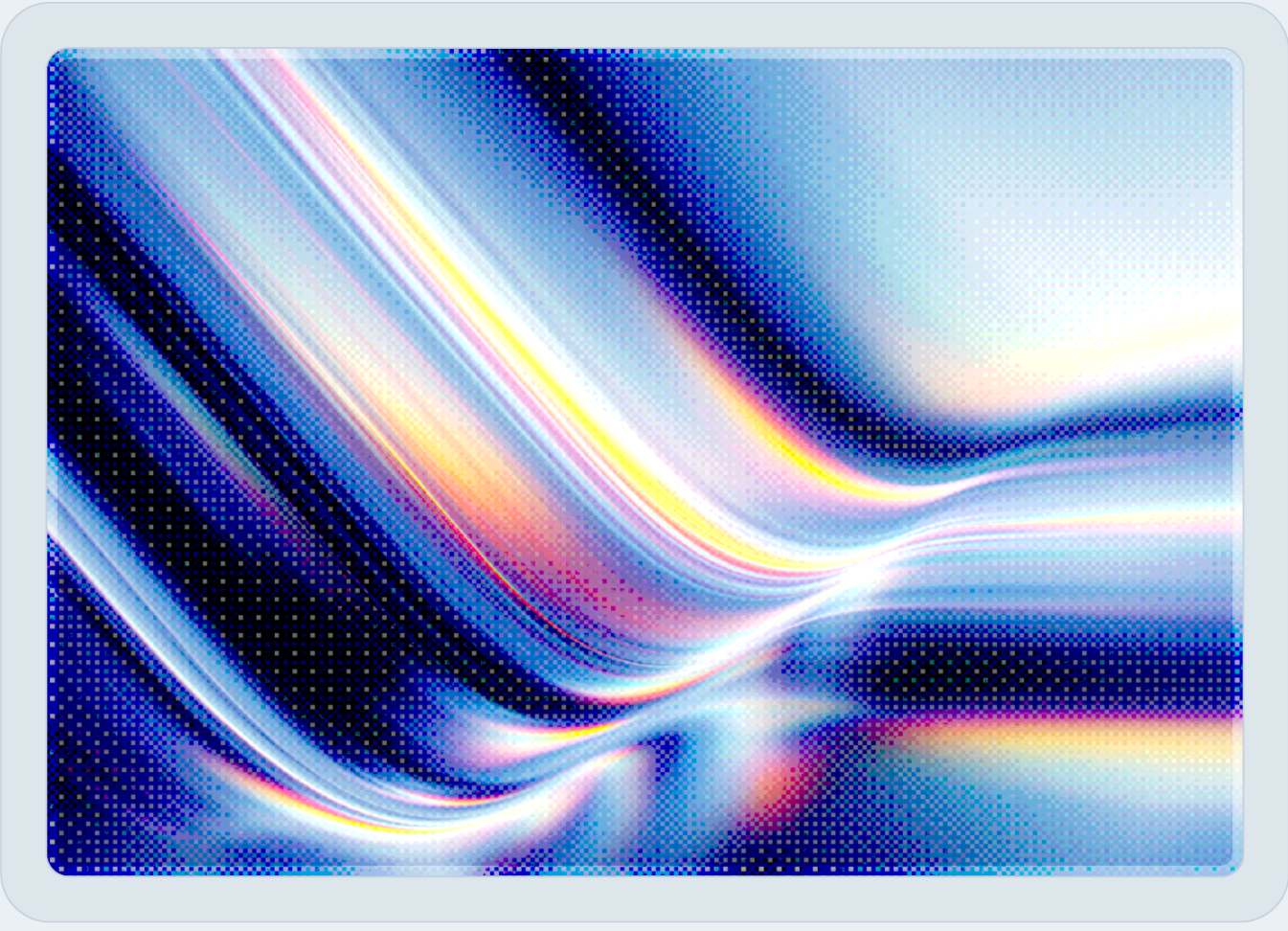
CSAI Foundation | Cloud Security Alliance

Miasma: Cross-Registry Supply Chain Credential Harvesting

Lessons from a Self-Propagating Worm Spanning npm, Go, and GitHub Actions

2026-06-27

AI-assisted Rapid Research



CSAI

CSA cloud security alliance®

Key Takeaways

- The Miasma campaign, a descendant of the Mini Shai-Hulud worm toolkit, compromised more than 110 npm packages across three coordinated waves in June 2026, affecting an estimated 80,000 to 117,000 weekly downloads and extending into the Go module ecosystem [1][2][3].
- Attackers weaponized a seven-week-old infostealer-harvested credential to initiate the campaign, illustrating how a single dormant developer secret can become a supply chain entry point weeks after initial compromise [5].
- The "Phantom Gyp" technique exploits npm's automatic native-build invocation via `binding.gyp` to execute arbitrary code at install time without declaring any lifecycle scripts, evading standard preinstall-hook monitoring [4][6].
- SLSA Build Level 3 provenance attestations were forged using legitimately obtained Sigstore OIDC tokens, allowing malicious packages to carry cryptographically valid supply-chain provenance records [3][4].
- Credential collection spans cloud providers (AWS, Azure, GCP), CI/CD platforms (GitHub, CircleCI, HashiCorp Vault, Kubernetes), and developer-local secrets (SSH keys, browser data, `.env` files, AI coding-assistant configuration) [1][2].
- AI coding-tool configuration files — including `.claude/settings.json`, `.cursor/rules/`, and VS Code task definitions — are both persistence vectors and lateral-movement targets, extending the campaign's reach into agentic developer workflows [2][4][6].

Background

Threat intelligence analysts have documented an evolution in supply chain attacks targeting open-source package registries — from opportunistic typosquatting toward coordinated, self-propagating campaigns that treat the developer credential ecosystem as infrastructure. The Miasma campaign, first observed on June 1, 2026, exemplifies this trajectory. Security analysts at SafeDep and StepSecurity describe Miasma as among the most technically layered supply chain campaigns publicly documented, combining registry poisoning, CI/CD pipeline compromise, cross-ecosystem propagation, and AI developer-tool persistence into a single, largely automated toolkit [1][2][3].

Miasma derives from the Mini Shai-Hulud worm codebase, which was released publicly under an MIT license on May 12, 2026 [5]. The open-source release likely lowered the barrier for other operators and complicates attribution, as analysts at Tenable note, assessing that it remains unclear whether Miasma represents TeamPCP operating under a new brand or a distinct actor repurposing the released toolkit [5]. The campaign's three-wave structure over five days suggests deliberate operational planning, as each wave introduced new techniques and expanded scope rather than repeating the same method.

The attack surface Miasma exploits reflects structural features of modern software development — shared credential graphs, automated publishing pipelines, and trusted configuration auto-execution — rather than a narrow vulnerability with a single patch. In large organizations, individual developers may maintain publishing credentials for tens or hundreds of packages, and CI/CD runners are commonly granted broad repository and cloud access to enable automated builds. AI coding assistants are granted read-write access to local file systems and remote repositories. When a high-privilege credential in that graph is compromised — such as a package-publishing token or a CI/CD runner token with broad repository access — much of the connected infrastructure can become reachable, as Miasma demonstrates. The distinguishing design feature of the toolkit is the degree to which it automates the traversal of that graph — harvesting credentials, republishing modified packages, and seeding persistence hooks across registries, repositories, and IDE configurations without human operator intervention [2][4][6].

The campaign's timeline also illustrates a threat-intelligence gap that defenders routinely underestimate. The compromised Red Hat employee credential that initiated Wave 1 had been present in infostealer aggregator logs for approximately seven weeks before attackers weaponized it on June 1 [5]. That dormancy period represents a window during which continuous credential-exposure monitoring could have detected and remediated the exposure before any package was touched. Few organizations outside of large enterprises currently operate continuous dark-web credential monitoring, and in this case the credential sat undetected until it was used.

Security Analysis

Campaign Timeline and Scope

The Miasma campaign unfolded in three waves over five days, each expanding the scope and refining the techniques employed. Wave 1, on June 1, 2026, compromised 32 packages in the `@redhat-cloud-services` npm scope across more than 90 versions, affecting packages with a combined 80,000 to 117,000 weekly downloads [1][5]. Microsoft's investigation confirmed that the compromise was

introduced through the legitimate `RedHatInsights/javascript-clients` CI/CD pipeline, where an attacker-controlled GitHub Actions OIDC publishing workflow published trojanized packages under Red Hat's trusted namespace [1].

Wave 2, on June 3, 2026, targeted 57 additional packages across independent namespaces, introducing the "Phantom Gyp" execution technique described below [4]. Wave 3, on June 5, extended the compromise to Microsoft Azure repositories; GitHub's trust and safety team disabled 73 repositories associated with the campaign [5]. Socket researchers subsequently identified a related cluster of LeoPlatform and RStreams packages carrying the Miasma payload family, reported in late June [3], suggesting the campaign continued to propagate beyond the initial three waves. The rapid progression from a single compromised credential to hundreds of repository actions reflects the automated propagation engine embedded in the toolkit.

The Phantom Gyp Execution Technique

The most-analyzed technical technique introduced in Wave 2 was a lifecycle-hook evasion method that security researchers at StepSecurity named "Phantom Gyp" [4]. Security guidance from StepSecurity notes that common monitoring approaches focus on declared lifecycle scripts (preinstall, postinstall) in `package.json`, leaving `binding.gyp`-triggered builds undetected by tools that audit only `package.json` entries [4]. Phantom Gyp bypasses declared-script monitoring entirely by adding a 157-byte `binding.gyp` file to a package. npm automatically invokes `node-gyp` when that file is present, treating it as a signal that the package requires native compilation. Within `binding.gyp`, the attacker embedded a gyp command-substitution expression that executes an arbitrary shell command during the build step. Because no preinstall or postinstall entry appears in `package.json`, tools that audit lifecycle scripts report nothing unusual [4][6].

Once execution was achieved, the payload progressed through a four-stage obfuscation pipeline designed to defeat static analysis. An outer ROT-N Caesar cipher shell decoded to reveal AES-128-GCM encrypted content, which in turn staged a Bun runtime loader, which finally executed an obfuscator.io-protected main payload with PBKDF2-HMAC-SHA-256 (200,000 iterations) string protection applied to sensitive values [1][2]. Per-build random encryption keys ensured that each republished package variant carried a unique binary signature, complicating hash-based detection [2].

Credential Harvesting and Exfiltration

The fully decoded payload executed a systematic credential sweep across the developer's local environment and any cloud or CI/CD resources accessible from it. The following table summarizes the credential categories targeted and the specific mechanisms employed:

Credential Category	Targets	Harvest Mechanism
Cloud providers	AWS IAM, Azure IMDS OAuth2, GCP service-account tokens	Cloud metadata endpoints (169.254.169.254, 169.254.170.2)
CI/CD platforms	GitHub tokens, CircleCI, HashiCorp Vault, Kubernetes service-account tokens	API enumeration, config file parsing, Vault auth paths
Developer secrets	npm tokens, PyPI tokens, SSH keys, GPG keys, Docker auth	File system traversal of known credential paths
Local environment	`.env` files, browser credentials, wallet data, shell history	Recursive file search with pattern matching
GitHub Actions runner	Masked workflow secrets	`/proc/<pid>/mem` read on the Runner.Worker process [4]
Password managers	1Password, Bitwarden, gopass, pass	Integration API calls and local data directory access
AI coding tools	`.claude/settings.json`, `.cursor/rules/`, VS Code tasks	Configuration file read; also written for persistence

GitHub Actions runner memory scraping deserves particular attention because it circumvents GitHub's secret-masking feature. The malware escalated to root via `sudo python3`, then read `/proc/2771/mem` (the Runner.Worker process) and applied a grep pattern that matched the unmasked form of otherwise-redacted workflow secrets [4]. Visual masking of secrets in workflow logs is insufficient as a confidentiality control against this technique. Privilege isolation of the runner environment is the most direct mitigation; complementary controls include ephemeral runner configurations that terminate immediately after job completion, `/proc` filesystem access restrictions via seccomp or AppArmor profiles, and disabling unrestricted `sudo` within the runner environment.

Exfiltration used three independent C2 channels implemented entirely through attacker-controlled GitHub repositories, avoiding the network-layer signatures associated with traditional C2 infrastructure [2]. Encrypted credential blobs were uploaded as repository artifacts; a persistent Python monitor on

victim machines polled GitHub hourly for new instructions, verified by RSA-PSS signature, and executed delivered scripts locally [2].

Self-Propagation and SLSA Provenance Forgery

After harvesting npm tokens, the malware enumerated every package maintained by the compromised account and republished modified versions of each one, effectively weaponizing the victim's publishing footprint [4][6]. The republished packages carried valid SLSA v1 provenance attestations generated through Sigstore's Fulcio certificate authority using legitimately obtained OIDC tokens from the victim's GitHub Actions environment [3][4]. Per the SLSA specification, Build Level 3 provenance attests that an artifact was built by a specific, hardened build platform from a verified source — making these attestations a widely used signal of supply chain integrity that many package consumers treat as high-confidence evidence. Miasma's forgery of these attestations demonstrates that SLSA provenance is only as trustworthy as the OIDC token issuance process it relies upon — a compromised CI/CD pipeline can produce cryptographically valid provenance for a malicious artifact.

Beyond npm, the same propagation engine injected malicious payloads into GitHub Actions workflow files by force-pushing trojanized orphan commits to semver-tagged releases while cloning the original commit metadata [2]. Any downstream workflow referencing `uses: owner/action@v1` would execute the malicious version without a version-pinning check flagging the change. The toolkit also contained templates for cross-registry propagation to RubyGems via `extconf.rb`, and PyPI typosquatting modules with `-mcp` and `-mpc` suffixes targeting the growing naming convention associated with Model Context Protocol packages [2].

Go Ecosystem and AI Coding-Tool Vectors

Socket researchers identified the same Miasma payload family embedded in a source archive for the Verana Blockchain Go module (`github.com/verana-labs/verana-blockchain@v0.10.1-dev.20`) [3]. Unlike npm packages, Go modules do not have a native install-hook mechanism; the compromise instead staged the payload at `.claude/index.js` within the source repository and configured a VS Code folder-open task (`node .claude/setup.mjs`) to execute it whenever a developer opened the project in their IDE [3]. Socket's AI scanner flagged the archive as known malware with high confidence based on the decode-and-eval loader pattern [3].

This Go module compromise highlights a second-order attack surface that affects any language ecosystem: the repository itself, rather than the registry artifact, can be the delivery vehicle when developer tooling automatically executes configuration files on project open. The `.claude/` directory

path — associated with the Claude AI coding assistant — appeared in both the Verana Blockchain compromise and in malicious packages injected into victim repositories by the propagation engine across the broader campaign [2][4]. Organizations using AI coding assistants that auto-load project configuration should treat configuration directories in unfamiliar repositories as potential execution triggers.

Persistence and Dead-Man Switch

The malware established persistence through multiple channels simultaneously. On developer machines it committed backdoor configuration files (`.claude/setup.mjs`, `.cursor/rules/setup.mdc`, `.vscode/tasks.json`) to repositories using stolen tokens, ensuring that any developer who subsequently opened the project in a supported IDE would re-execute the payload [4]. The persistent Python monitor (`GITHUB_MONITOR.py`) ran as a background process, polling GitHub hourly for operator-delivered scripts with a 72-hour self-termination timer [2].

The toolkit embedded a destructive dead-man switch: if a victim revoked the monitored personal access token, the payload executed `rm -rf ~/; rm -rf ~/Documents` before terminating [2]. The campaign marker "RevokeAndItGoesKaboom" embedded in the toolkit's documentation makes the coercive intent explicit. The dead-man switch is designed to deter immediate credential rotation — which is the standard first response to a supply chain compromise — by making that action destructive, and should be treated as a critical operational consideration when planning incident response.

Recommendations

Immediate Actions

Organizations that installed `@redhat-cloud-services` packages between June 1 and June 3, 2026, or any LeoPlatform, RStreams, or other packages identified as part of the campaign through late June 2026 [3][4], should treat their environments as potentially compromised. The priority sequence is: isolate affected systems from sensitive network resources, then rotate credentials in a specific order — cloud provider credentials first, followed by GitHub tokens, npm publish tokens, and Kubernetes service-account tokens — before rotating the locally stored npm and SSH keys that the malware would use to expand its footprint. Rotating the GitHub personal access token before securing the runner environment may trigger the destructive dead-man switch; responders should disable the persistent monitor process (`GITHUB_MONITOR.py`) before token rotation if it is present [2].

Search repositories for injected AI coding-assistant configuration files at `.claude/setup.mjs`, `.cursor/rules/setup.mdc`, `.vscode/tasks.json`, and `.github/setup.js`. These files may persist even after the originating package is removed. Review GitHub audit logs and CloudTrail for unauthorized API activity, unexpected repository creation, and artifact upload events in the timeframes corresponding to each campaign wave.

Short-Term Mitigations

The Phantom Gyp evasion demonstrates that monitoring declared lifecycle scripts alone is insufficient for npm security. Development teams should run `npm install --ignore-scripts` in CI/CD environments unless native compilation is explicitly required by the specific packages in use; this flag prevents both preinstall hooks and `node-gyp` invocations [4]. Lockfiles with cryptographic integrity hashes provide a complementary control: the package manager will detect a content mismatch before executing any install-time code. Development and security teams should flag any multi-megabyte root `index.js` files that are not the declared package entry point, and monitor for publish bursts — multiple package versions from a single maintainer account within a short window — as an indicator of automated self-propagation [4].

GitHub Actions workflow security requires specific hardening beyond the standard recommendations. OIDC token scopes should be restricted to the minimum necessary branch patterns, and `id-token: write` permissions should not be granted to workflows that do not explicitly require signed provenance generation [2][5]. Workflows that publish packages should require pull-request-based approval gates or two-person review on the triggering branch, preventing a single compromised token from initiating an automated publish cycle. Actions references should be pinned by commit SHA rather than by semver tag, because tag-based references are vulnerable to the force-push hijacking technique Miasma employed [2].

Organizations deploying AI coding assistants should establish policy governing which configuration directories those tools are permitted to auto-execute on project open. The `.claude/`, `.cursor/`, and `.vscode/tasks.json` vectors are not exclusive to any single assistant; they reflect a general pattern in which developer tooling trusts project-local configuration files. Sandboxing the execution context of assistant-invoked setup scripts, or requiring explicit user approval before executing project configuration on first open, would substantially reduce the attack surface these vectors represent.

Strategic Considerations

The seven-week credential dormancy that enabled Wave 1 represents a detection gap that cannot be closed by endpoint or network controls alone. Stolen credentials that appear in infostealer aggregator logs remain invisible to every tool that watches only the organization's own perimeter. Continuous exposure monitoring — services that scan underground credential markets for developer account credentials belonging to an organization's domains — provides important coverage for the credential dormancy gap that endpoint and network controls cannot address [5]. Complementary controls that shorten the usable life of a stolen credential include FIDO2 authentication, short-lived OIDC tokens, and mandatory periodic rotation policies. For most organizations, the operational cost of continuous credential monitoring is modest relative to the potential impact of a supply chain compromise affecting thousands of downstream consumers.

The SLSA provenance forgery observed in Miasma does not invalidate SLSA as a framework; it identifies a specific assumption that SLSA's trust model makes explicit but that is easy to miss in practice: SLSA Level 3 attests nothing about the trustworthiness of the build environment's OIDC identity provider. SLSA Build Level 3 provenance attests that an artifact was built by a specific build platform in an isolated environment, but only if the OIDC token used to generate that attestation came from a trustworthy execution context. When a CI/CD pipeline is itself compromised, the resulting attestation is cryptographically valid but factually false. Defenders should treat provenance verification as a necessary but not sufficient control, and pair it with monitoring for anomalous publish patterns and unexpected attestation-generation events.

The public release of the Mini Shai-Hulud toolkit under a permissive open-source license meaningfully expands the population of threat actors capable of executing supply chain campaigns of this complexity. Capabilities that previously required specialized development effort — multi-registry propagation engines, OIDC token extraction from runner process memory, automated SLSA provenance forgery — are now available as a deployable open-source codebase, meaningfully lowering the barrier for technically capable but less-resourced threat actors. Security teams should update their threat models for software supply chain attacks accordingly.

CSA Resource Alignment

The Miasma campaign maps directly onto several CSA frameworks that provide actionable guidance for the vulnerabilities it exploits.

The CSA Cloud Controls Matrix v4.1 Supply Chain Management, Transparency, and Accountability (STA) domain contains sixteen control specifications addressing third-party and CI/CD pipeline security [7]. The Miasma attack chain exploited weaknesses in at least three STA control areas: token lifecycle management, pipeline access controls, and artifact integrity verification. Organizations using CCM for cloud security assessments should treat the STA domain controls as directly relevant to package-registry publishing pipelines, not only to traditional vendor relationships.

The CSA DevSecOps Automation guidance, part of the Six Pillars of DevSecOps series, addresses pipeline security and the integration of security controls into automated workflows [8]. The CI/CD exploitation techniques in Miasma — OIDC token theft, environment-protection-rule bypass, and automated package republication — are precisely the scenarios that DevSecOps automation controls are designed to prevent. In particular, the guidance's emphasis on least-privilege service accounts and mandatory approval gates for deployment actions would have materially constrained the automated propagation the campaign achieved.

The Agentic Trust Framework (ATF, v0.9.1, April 2026), stewarded by the CSAI Foundation and authored by Josh Woodruff / MassiveScale.AI [9], addresses the trust model for autonomous agents operating in developer environments. Miasma's exploitation of AI coding-assistant configuration directories — staging payloads in `.claude/`, `.cursor/`, and VS Code task files — is precisely the threat class ATF addresses through its identity, behavior, and segmentation elements. ATF's four-level maturity model (Intern → Junior → Senior → Principal) applies directly to the access grants that AI coding assistants currently receive: an assistant with `Principal`-level file-system and network access in an unconstrained project context is a high-value compromise target. Restricting AI coding assistants to narrower access scopes and implementing explicit approval gates for configuration-file execution would reduce the attack surface Miasma leveraged.

MAESTRO, CSA's threat-modeling framework for multi-agent AI systems, provides a structured approach to identifying where AI agent configurations become attack surfaces [10]. The Miasma campaign's discovery that VS Code tasks, Claude project configurations, and Cursor rules can serve as execution vectors for arbitrary JavaScript points to a threat category MAESTRO practitioners should include in their agent attack-surface inventories: trusted configuration files in developer environments that are auto-executed without integrity verification.

References

[1] Microsoft Security Blog. "Preinstall to persistence: Inside the Red Hat npm Miasma credential-stealing campaign." Microsoft, June 2, 2026.

[2] SafeDep. "Inside the Miasma Software Supply Chain Attack Toolkit." SafeDep, June 2026.

[3] Socket. "Miasma Mini Shai-Hulud Hits LeoPlatform npm Packages and Go Ecosystem." Socket.dev, June 25, 2026.

[4] StepSecurity. "Miasma npm Supply Chain Attack: Self-Spreading Worm via Phantom Gyp." StepSecurity, June 3, 2026.

[5] Tenable. "What does the Miasma worm campaign reveal about supply chain threats?" Tenable, June 23, 2026.

[6] Upwind. "Miasma npm Worm: RedHat Supply Chain Credential Theft." Upwind Security, June 4, 2026.

[7] Cloud Security Alliance. "Cloud Controls Matrix v4.1." CSA, 2021 (current version).

[8] Cloud Security Alliance. "The Six Pillars of DevSecOps: Automation." CSA DevSecOps Working Group, 2020.

[9] Woodruff, Josh / MassiveScale.AI; CSAI Foundation. "Agentic Trust Framework v0.9.1." Cloud Security Alliance, April 2026.

[10] Cloud Security Alliance AI Safety Initiative. "MAESTRO: Multi-Agent Environment, Security, Threat, and Risk Observatory." CSA Labs, 2025.

