INTELLIGENCE BRIEFING
Security Command Center
TLP:CLEAR
2026-03-29 18:42 UTC

GlassWorm Moves Deeper: Dependency-Layer Implants Expose Gaps in Extension-Focused Scanning

THREAT CAMPAIGN | **HIGH** | CVSS 7.5

SCC Item ID	SCC-CAM-2026-0040
Type	Threat Campaign
Severity	HIGH
CVSS Base Score	7.5
Affected Products	Software dependency ecosystems, specific package managers and vendors not confirmed in available source material; dependency supply chain broadly
Published	2026-03-17

Executive Summary

The GlassWorm malware family has shifted tactics, embedding malicious code inside software dependencies rather than browser or application extensions, a layer most security scanning tools do not reach. Dozens of confirmed malicious packages have been identified across software dependency ecosystems, indicating a deliberate, ongoing campaign rather than isolated incidents. Organizations that rely on extension-focused scanning alone have a structural blind spot in their software supply chain, increasing risk of undetected compromise in development pipelines and production environments.

Technical Analysis

GlassWorm represents an evolved supply chain threat that has migrated from extension-layer implants to dependency-layer implants, exploiting a well-documented gap in security tooling coverage. Confirmed malicious packages number in the dozens; specific package managers and registries are not confirmed in available source material. No CVE has been assigned. Relevant weaknesses: CWE-829 (inclusion of functionality from untrusted control sphere), CWE-494 (download of code without integrity check), CWE-693 (protection mechanism failure enabling evasion). MITRE ATT&CK techniques observed or attributed: T1195.001 (Supply Chain Compromise: Compromise Software Dependencies and Development Tools), T1554 (Compromise Client Software Binary), T1072 (Software Deployment Tools), T1036 (Masquerading), T1027 (Obfuscated Files or Information). No patch is applicable, this is a campaign-level supply chain threat, not a single patchable vulnerability. Remediation requires process and tooling changes at the dependency verification layer. Attribution to a named threat actor is not confirmed in available source material. Source quality is assessed at 0.4; primary sources are T3. Technical specifics should be treated as directionally accurate pending higher-tier confirmation.

Action Checklist

1. Step 1, Immediate: Audit current scanning coverage to determine whether your SAST, SCA, or registry scanning tools inspect transitive and direct dependencies, not only extensions or top-level packages. Identify the gap before assessing exposure.
2. Step 2, Detection: Review dependency manifests (package.json, requirements.txt, go.mod, pom.xml, etc.) for recently added or modified packages. Cross-reference package names and versions against known-good baselines and public malicious package databases (e.g., OpenSSF Package Analysis, Socket.dev, Phylum).
3. Step 3, Assessment: Inventory all build pipelines, CI/CD systems, and developer environments for dependency resolution practices. Flag any pipeline that pulls dependencies at build time without hash-pinning or integrity verification (CWE-494). Prioritize internet-facing and production-adjacent pipelines.
4. Step 4, Communication: Notify development leads and DevSecOps teams of the dependency-layer blind spot. If your organization uses third-party software vendors, request confirmation that their dependency scanning covers this attack surface. Escalate to CISO if pipeline inventory reveals unverified dependency resolution in production paths.
5. Step 5, Long-term: Enforce dependency pinning with cryptographic hash verification across all package managers. Implement a private registry or artifact proxy (e.g., Artifactory, Nexus) with allow-listing to reduce exposure to public registry poisoning. Map controls to NIST SP 800-161 (Supply Chain Risk Management) and CISA Secure Software Development guidance. Review and update software composition analysis tooling to include transitive dependency inspection.

IR / Forensic Enrichment

Triage Priority	IMMEDIATE
Escalation Criteria	Escalate to CISO immediately if audit reveals production CI/CD pipelines pulling dependencies without hash-pinning or if any confirmed malicious package from public threat feeds has been resolved in a deployed application within the last 180 days.
Recovery Notes	Post-containment recovery: (1) Force re-build all affected applications with pinned, verified dependencies and updated SCA scanning enabled. (2) Conduct retrospective dependency audit for the last 12 months of releases to identify exposure window for GlassWorm or similar threats; generate timeline of when malicious packages (if any) were resolved. (3) Update incident response playbook with dependency-supply-chain scenarios, including detection rules for abnormal package changes, CI/CD pipeline monitoring for hash verification failures, and vendor communication templates for SCA confirmation.

Forensic Artifacts	Git commit history and diffs for all dependency manifests (package.json, requirements.txt, go.mod, pom.xml) — git log --all --oneline --full-history -- '**/package*.json' '**/requirements*.txt' '**/go.mod' '**/pom.xml' | Dependency lock files with timestamps (package-lock.json, Pipfile.lock, go.sum, pom.lock, maven-lock.json) — preserve original timestamps via ls -la and stat commands | CI/CD pipeline logs and build artifacts including dependency resolution steps, hash verification outputs, and package registry queries — export from Jenkins, GitHub Actions, GitLab CI, CircleCI audit logs | Package registry API responses and download metadata (tarball hashes, publish dates, publisher identity) retrieved from npm, PyPI, Maven Central, Go proxy — save full API responses with timestamps | SCA/SAST tool scan reports and configuration files showing coverage gaps (.snyk.yml, sonarqube.properties, .trivyignore, dependency-check configuration) — capture before and after remediation

Per-Action IR Details

Step 1, Immediate: Audit current scanning coverage to determine whether your SAST, SCA, or registry scanning tools inspect transitive and direct dependencies, not only extensions or top-level packages. Identify the gap before assessing exposure.

NIST Phase: Preparation

Reference: NIST 800-61r3 §2.1 (Preparation phase — detection capabilities assessment)

Controls: NIST SI-4 (Information System Monitoring), NIST RA-3 (Risk Assessment), CIS 6.2 (Ensure Software is Inventoried)

Compensating: No enterprise SCA? Use free tools: (1) npm audit (Node.js), (2) pip-audit (Python), (3) Trivy by Aqua Security (container + dependency scanning, free CLI), (4) OWASP Dependency-Check (free, CLI-based). Cross-check findings against OpenSSF Package Analysis API (free). Document tool coverage in a spreadsheet: [package manager] → [tool] → [inspects transitive deps: Y/N] → [last run date].

Evidence: Before auditing, capture baseline: (1) List all active SAST/SCA tools with configuration files (e.g., .snyk.yml, sonarqube.properties), (2) screenshot or export current scan reports with timestamp, (3) document all CI/CD pipeline steps that invoke dependency resolution, (4) retrieve manifest file versions from Git history (git log --oneline -- package.json) for last 90 days to establish change velocity baseline.

Step 2, Detection: Review dependency manifests (package.json, requirements.txt, go.mod, pom.xml, etc.) for recently added or modified packages. Cross-reference package names and versions against known-good baselines and public malicious package databases (e.g., OpenSSF Package Analysis, Socket.dev, Phylum).

NIST Phase: Detection Analysis

Reference: NIST 800-61r3 §3.2 (Detection and Analysis — analysis techniques)

Controls: NIST SI-4 (Information System Monitoring), NIST SA-3 (System Development Life Cycle), CIS 6.2 (Software Inventory), CIS 6.3 (Address Unauthorized Software)

Compensating: Manual manifest review: (1) Extract all manifests from Git: git log --full-history --all -- '**/package.json' '**/requirements.txt' '**/go.mod' | parse versions, (2) for each package added/modified in last 30 days, query Socket.dev API (free tier, 100 requests/month) or run local OWASP Dependency-Check with --enableExperimental flag against lock files, (3) compare hashes using: npm view [package]@[version] dist.tarball | sha256sum, (4) search package names in public vulnerability databases (NVD, GitHub Security Advisories) manually.

Evidence: Capture before scanning: (1) Export full dependency lock files (package-lock.json, Pipfile.lock, go.sum, pom.lock) with timestamps, (2) Git commit history for manifests: git log --oneline --all -- '**/package.json' -- '**/requirements.txt' > dep_changes.txt, (3) list of installed packages at build time: npm ls --all > baseline_deps.txt (or equivalent for each manager), (4) download hashes from package registries before querying threat feeds (save to file for chain-of-custody), (5) document any custom/internal packages: grep -r 'registry' .npmrc ~/.npmrc pom.xml -- save configuration.

Step 3, Assessment: Inventory all build pipelines, CI/CD systems, and developer environments for dependency resolution practices. Flag any pipeline that pulls dependencies at build time without hash-pinning or integrity verification (CWE-494). Prioritize internet-facing and production-adjacent pipelines.

NIST Phase: Detection Analysis

Reference: NIST 800-61r3 §3.2.5 (Determining scope and impact)

Controls: NIST SA-3 (System Development Life Cycle), NIST SA-12 (Supply Chain Protection), NIST CM-5 (Access Restrictions for Change), CIS 8.4 (Restrict Library Permissions)

Compensating: No enterprise inventory tool? Manual audit: (1) For each CI/CD platform (GitHub Actions, GitLab CI, Jenkins, CircleCI), export all pipeline definitions: GitHub: gh api repos/[org]/[repo]/contents/.github/workflows --recursive, Jenkins: Jenkins XML API /api/json, (2) grep for dependency installation commands: 'npm install', 'pip install', 'go get', 'mvn install', 'gradle build' without corresponding hash-pinning (grep for 'package-lock.json --frozen-lockfile' or '--require-hashes'), (3) for each flagged pipeline, check if lock files are committed to repo: git ls-files | grep -E 'lock|freeze', (4) document developer machine setup: retrieve .bashrc, .zshrc, conda environment.yml, .gradle/gradle.properties from team members (with consent) to identify uncontrolled resolution, (5) create a matrix: [Pipeline] → [Dependency Manager] → [Hash-Pinned: Y/N] → [Production-Adjacent: Y/N] → [Last Modified Date].

Step 4, Communication: Notify development leads and DevSecOps teams of the dependency-layer blind spot. If your organization uses third-party software vendors, request confirmation that their dependency scanning covers this attack surface. Escalate to CISO if pipeline inventory reveals unverified dependency resolution in production paths.

NIST Phase: Containment

Reference: NIST 800-61r3 §3.3 (Containment — communication and coordination)

Controls: NIST IR-1 (Incident Response Policy), NIST IR-4 (Incident Handling), NIST CA-7 (Continuous Monitoring), CIS 2.1 (Ensure Explicit Incident Response Team)

Compensating: No formal IR process? Create immediate notification protocol: (1) Draft brief technical summary (1 page max): GlassWorm threat, dependency-layer attack vector, your organization's gap (no hash-pinning in pipeline [X], no SCA for transitive deps in [Y]). Include: threat source (e.g., 'CISA alert'), CVE NA/context, CVSS 7.5, immediate actions. (2) Distribute to: all development team leads via email with security team CC, DevOps/SRE leads, architecture review board. (3) Request vendor SCA confirmation in writing: 'Does your tool inspect transitive dependencies? Are hashes verified at pull time? Which package managers?' Save responses to audit trail. (4) Escalation trigger: If any production pipeline lacks hash-pinning, send 24-hour escalation memo to CTO/CISO with subject 'SUPPLY CHAIN RISK - UNVERIFIED DEPENDENCY RESOLUTION IN PROD' listing affected pipelines.

Evidence: Capture before notifying: (1) Date/time of threat publication (CISA advisory, news source), (2) screenshot of current SCA tool configuration showing coverage gaps, (3) copy of pipeline definitions with hash-pinning status (from Step 3 output), (4) list of active vendors/third-party SaaS tools with SCA claims, (5) internal incident ID or ticket number for tracking, (6) log all outbound communications: save email thread, Slack thread exports, meeting notes with timestamps.

Step 5, Long-term: Enforce dependency pinning with cryptographic hash verification across all package managers. Implement a private registry or artifact proxy (e.g., Artifactory, Nexus) with allow-listing to reduce exposure to public registry poisoning. Map controls to NIST SP 800-161 (Supply Chain Risk Management) and CISA Secure Software Development guidance. Review and update software composition analysis tooling to include transitive dependency inspection.

NIST Phase: Recovery

Reference: NIST 800-61r3 §3.4 (Recovery) and NIST 800-161 §3 (Supply Chain Risk Management Practices)

Controls: NIST SA-3 (System Development Life Cycle), NIST SA-10 (Developer Configuration Management), NIST SA-12 (Supply Chain Protection), NIST CM-5 (Access Restrictions for Change), CIS 8.3 (Address Unauthorized Software), CIS 8.4 (Restrict Library Permissions)

Compensating: Budget-constrained environment? Implement in phases: Phase 1 (Weeks 1-2): Enforce lock file commits — add Git pre-commit hook: if ! git diff --cached | grep -q 'package-lock.json\|Pipfile.lock\|go.sum'; then echo

'Lock file must be committed'; exit 1; fi. Add CI/CD check: npm ci --frozen-lockfile, pip install --require-hashes, go mod verify. Phase 2 (Weeks 3-8): Private registry — use Nexus OSS (free) or Artifactory Free tier; configure npm/pip/Maven to proxy public registries through private instance with allow-list. Document in CONTRIBUTING.md. Phase 3 (Weeks 9-12): Update SCA — swap free tier tool for Trivy + OWASP Dependency-Check + Socket.dev API (free) in CI pipeline; configure to fail build on high-risk findings. Phase 4 (Ongoing): Map to NIST 800-161 controls in your security documentation and audit responses.

Evidence: Capture post-recovery to validate: (1) Compare all lock files before/after pinning implementation: git diff [old-commit] [new-commit] -- '*lock*', (2) audit CI/CD logs showing hash verification success: grep -i 'verify\|hash\|integrity' [build logs] with timestamps, (3) document Artifactory/Nexus configuration: export allow-list config, proxy rules, retention policies, (4) retrieve SCA tool scan reports post-update showing transitive dependency analysis, (5) create policy document: 'Dependency Pinning and Verification Policy' with effective date, version history, team sign-offs.

Detection Guidance

No confirmed IOCs (hashes, domains, IPs) are available in the source material at the time of this report. Detection should focus on behavioral and structural indicators. Check CI/CD logs for dependency resolution events that pull packages not present in a verified lockfile. Look for packages with names closely resembling internal or popular public packages (typosquatting pattern associated with T1036). Flag packages published recently by unknown maintainers that appear in dependency trees of sensitive projects. In SIEM or pipeline logging, query for install or build events that bypass lockfile enforcement (e.g., npm install without --frozen-lockfile, pip install without hash checking). For environments with SCA tooling, run a scan specifically targeting transitive dependencies and compare results against your last known-clean baseline. Any package flagged by OpenSSF Package Analysis, Phylum, or Socket.dev as suspicious should be treated as high-priority for manual review. Note: absence of confirmed IOCs reflects current source limitations, not absence of indicators. Check threat intelligence feeds and vendor advisories for GlassWorm-associated package hashes as reporting matures.

Indicators of Compromise

Type	Value	Context	Confidence
HASH	`[not available in source material]`	No confirmed package hashes or file hashes attributed to GlassWorm dependency-layer implants are present in available T3 sources. Monitor public malicious package databases for emerging IOCs.	LOW

Framework Mappings

MITRE-ATTACK

- **T1036** — Masquerading
- **T1072** — Software Deployment Tools
- **T1554** — Compromise Host Software Binary
- **T1195.001** — Compromise Software Dependencies and Development Tools

- **T1027** — Obfuscated Files or Information

NIST-800-53R5

- **SI-3** — Malicious Code Protection
- **SI-4** — System Monitoring
- **SI-7** — Software, Firmware, and Information Integrity
- **CM-3** — Configuration Change Control

OWASP-TOP10-2021

- **A08:2021** — Software and Data Integrity Failures

CIS-V8

- **2.5**
- **2.6**

MITRE ATT&CK Mapping

Technique ID	Technique Name	Tactic
T1036	Masquerading	Defense-Evasion
T1072	Software Deployment Tools	Execution
T1554	Compromise Host Software Binary	Persistence
T1195.001	Compromise Software Dependencies and Development Tools	Initial-Access
T1027	Obfuscated Files or Information	Defense-Evasion

Sources

Source	URL	Tier
Security News	https://www.darkreading.com/application-security/glassworm-malware-...	T3
A dataset on vulnerabilities affecting dependencies in software ...	https://www.sciencedirect.com/science/article/pii/S2352340925006274	T3
Dependency Confusion: A Threat Actor in the Modern Software ...	https://medium.com/tom-tech/dependency-confusion-a-threat-actor-in-...	T3
Package Managers are Evil - Lobsters	https://lobste.rs/s/zvdtdn/package_managers_are_evil	T3

Source	URL	Tier
Your Security Scans Are Missing Critical Vulnerabilities—Here's Why	https://www.netrise.io/xiot-security-blog/why-scanners-miss-vulnera...	T3
Dependency Confusion: How I Hacked Into Apple, Microsoft and ...	https://medium.com/@alex.birsan/dependency-confusion-how-i-hacked-i...	T3
[PDF] Precisely Identify Affected Packages of Known Vulnerabilities in npm ...	https://www.ndss-symposium.org/wp-content/uploads/2026-f1902-paper.pdf	T3
CVE-2025-14010 - Red Hat Customer Portal	https://access.redhat.com/security/cve/cve-2025-14010	T3
Addressing the npm Manifest Confusion Vulnerability - JFrog	https://jfrog.com/blog/addressing-the-npm-manifest-confusion-vulner...	T3

Generated 2026-03-29 18:42 UTC by TJS Security Command Center