A Practitioner's Guide to Responding to the TeamPCP Supply Chain Attacks

Investigating, remediating, and hardening your environment in the wake of the TeamPCP campaign — from an organization that went through the process itself.

Devon Powley
Director of Information Security
Robert Haynes
Principal Technical Marketing Engineer

ENDOR LABS

Table of Contents

Introduction

The TeamPCP attacks have had security teams across the industry firing up yet another incident response effort. What began as a compromised GitHub Action in the Trivy repository has cascaded — in just six days — into a multi-vector campaign affecting npm packages, Python libraries, container images, IDE extensions, and CI/CD pipelines across thousands of organizations. One stolen token compromised five ecosystems.

The blast radius continues to expand. The initial GitHub Action credential-stealing attack spread to the popular LiteLLM Python package (3.6 million downloads/day), over 60 npm packages via the self-propagating CanisterWorm, Checkmarx's KICS GitHub Actions, and 44 defaced Aqua Security repositories. An Iran-targeted wiper component was also discovered. Given the volume of CI/CD pipelines compromised and the credentials harvested, it seems likely the attack is far from over.

As the SANS Cloud Security team noted in their analysis, this is not an isolated event — it represents an accelerating trend in supply chain attacks that has evolved from SolarWinds (2020, nation-state, 1-hop) through Codecov (2021) and tj-actions (2025, 4+ hops) to TeamPCP (2026, criminal group, 3+ hops plus a self-propagating worm). Each generation is faster, more automated, and harder to contain.

Endor Labs is no exception to this. As a security vendor that publishes binaries, CI/CD apps, and GitHub Actions, we are acutely aware of our responsibility to maintain the integrity of our environment and to rapidly investigate potential threats. We don't claim to be perfect or invulnerable, but we learn and improve with every iteration — even if a break from such frequent practice would be welcome.

This guide distills our response into a practical framework that other organizations can follow. It is not intended as a definitive playbook — every environment is different — but as a structured starting point informed by real-world experience.

Our response breaks down into four phases:

Investigation — Determine exposure and assess impact

Remediation — Contain the damage and restore integrity

Prevention — Harden defenses against future supply chain attacks

How the attack unfolded

Before diving into the response framework, it helps to understand the attack chain — both to appreciate the scope of what you're investigating and to recognize that a single misconfiguration can cascade far beyond your own environment.

Day 0 (February 28): Initial access

An AI-powered bot called **hackerbot-claw** — self-described as an "autonomous security research agent" — had been systematically probing thousands of public GitHub repositories for workflow misconfigurations. It found one in Trivy's `apidiff.yaml`: a `pull_request_target` trigger that ran with base repository secrets but checked out code from incoming pull requests. This is a well-documented anti-pattern — the same vector exploited in SpotBugs (2024) and Nx (2025).

The bot exploited the misconfiguration to extract the **aqua-bot Personal Access Token** — a single credential with write access to the entire Trivy ecosystem.

March 1: The critical mistake

Aqua Security detected the breach and rotated credentials. But the rotation was not atomic: the attacker was watching, and because new credentials were issued before the compromised credential was revoked, the attacker captured the refreshed tokens during the gap. As Itay Shakury, Aqua's VP of Open Source, later acknowledged: *"We rotated secrets and tokens, but the process wasn't atomic, and attackers may have been privy to refreshed tokens."*

This is a crucial lesson for any organization's incident response playbook. Routine credential rotation (issue new → migrate → verify → revoke old) is fundamentally different from incident-response rotation (revoke immediately → reissue out-of-band → deploy after access path eliminated). Rotating through compromised infrastructure gives the attacker a second chance.

Day 1 (March 19): Weaponization

TeamPCP force-pushed 76 of 77 `trivy-action` tags and all 7 `setup-trivy` tags to malicious commits. They also released a backdoored Trivy v0.69.4 binary, distributed via GitHub Releases, Docker Hub, GHCR, and Amazon ECR.

The payload — the "TeamPCP Cloud Stealer" — operated in two modes depending on the runner environment. On **GitHub-hosted runners**, it enumerated `Runner.Worker` and `Runner.Listener` processes, read `/proc/PID/mem` to extract secrets marked `isSecret:true`,

and used a base64-decoded Python script for memory scraping. On **self-hosted runners**, it swept 50+ filesystem paths for SSH keys, cloud credentials, Kubernetes tokens, Docker configs, database credentials, .env files, cryptocurrency wallets, TLS private keys, and git credentials.

Critically, **the malware then ran the real Trivy scan** — victims saw green checkmarks in their CI/CD pipelines while their credentials were being exfiltrated.

Data was encrypted with AES-256 + RSA-4096 and sent to a vendor-specific typosquatted C2 domain (`scan.aquasecurtiy[.]org`). A fallback mechanism created public `tpcp-docs` repositories on the victim's own GitHub account and uploaded the encrypted data as release assets.

Days 2–6: The cascade

- **Day 2 (March 20):** Stolen npm tokens were used to launch **CanisterWorm** — a self-propagating npm supply chain worm. It used `findNpmTokens()` to locate tokens, then called `deploy.js` to auto-increment version numbers and republish backdoored packages autonomously, without human intervention. 141 malicious artifacts across 66+ packages were published. The worm's C2 was hosted on the **Internet Computer Protocol blockchain** (a first in documented supply chain attacks), making it resistant to traditional takedowns and requiring DFINITY Foundation intervention.
- **Day 4 (March 22):** 44 Aqua Security repositories were defaced. An Iran-targeted wiper component was discovered (using timezone/locale checks). Malicious Docker images v0.69.5 and v0.69.6 were pushed to Docker Hub.
- **Day 5 (March 23):** Checkmarx was compromised via the `ast-phoenix` and `cx-plugins-releases` service accounts. All 35 KICS GitHub Action tags were rewritten. Malicious VS Code extensions were published to OpenVSX. A new C2 domain (`checkmarx.zone`) was introduced along with Kubernetes persistence code.
- **Day 6 (March 24):** LiteLLM was compromised on PyPI — a direct result of Trivy scanning in LiteLLM's CI/CD pipeline, which led to stolen PyPI tokens. Malicious versions 1.82.7 and 1.82.8 were published just 13 minutes apart. The package has 3.6 million downloads per day. Version 1.82.8 introduced a particularly aggressive `.pth` file persistence mechanism that executes on *any* Python startup, even if LiteLLM is never imported. A fork bomb bug in the malware was ironically the accidental detection mechanism that alerted users.
- **Day 9 (March 27)**: Two PyPI releases of telnyx were backdoored with a multi-stage attack that hides its payload inside WAV audio files, steals everything from SSH keys to Kubernetes secrets, and exfiltrates it all to a bare-IP C2 server. The same RSA key from the litellm compromise ties this directly to TeamPCP.

Phase 1: Investigation

The first priority is establishing whether your organization was affected by any stage of the TeamPCP campaign, and whether you were susceptible to the initial attack vectors.

The first priority is establishing whether your organization was affected by any stage of the TeamPCP campaign, and whether you were susceptible to the initial attack vectors.

Understanding the attack surface

This campaign is notable for its breadth — five ecosystems compromised from a single stolen token. The compromises span multiple artifact types:

- **GitHub Actions** — `trivy-action` (75 of 76 tags rewritten), `setup-trivy` (all 7 tags), Checkmarx KICS Actions (all 35 tags)
- **CLI binaries** — Trivy v0.69.4
- **Container images** — Trivy Docker Hub images v0.69.4, v0.69.5, v0.69.6, plus GHCR and ECR variants
- **Python packages** — LiteLLM v1.82.7 and v1.82.8, telynx v4.87.1 and v4.87.2 on PyPI
- **npm packages** — 60+ packages compromised via CanisterWorm self-propagating worm
- **Go libraries** — Potential exposure through Trivy's Go module ecosystem
- **VSCode extensions** — Checkmarx IDE extensions via OpenVSX

For each of these vectors, you need a thorough check of usage across every place you consume dependencies — not just your application code, but your CI/CD pipelines, container images, developer workstations, remote hosts, and other runtime environments where a library might have been installed outside the normal build process.

Searching for affected versions

Start with your AppSec scanning tools to identify known-affected versions in your codebase and published artifacts. But recognize that scanning alone won't catch everything. This is an all-hands exercise that should also leverage your EDR and CNAPP platforms to investigate exposure in local and runtime environments.

Affected versions to search for:

Component	Affected Versions	Known-Clean Version
Trivy binary	v0.69.4	v0.69.2, v0.69.3
trivy-action	v0.0.1 – v0.34.2 (all tags rewritten)	v0.35.0
setup-trivy	All versions (all tags rewritten)	Verify via SHA only
Trivy Docker images	v0.69.4, v0.69.5, v0.69.6	v0.69.3 and earlier
LiteLLM	v1.82.7, v1.82.8	v1.82.6
Checkmarx KICS Actions	All 35 tags rewritten	Verify via SHA only
npm (CanisterWorm)	60+ packages — see full IOC list	N/A — remove affected packages
telynx	v4.87.1, v4.87.2	v4.87.0

Searching for indicators of compromise

Beyond checking for affected versions, look for evidence of other characteristics of the malware in your environment.

Filesystem and process artifacts

- `/tmp/pglog` — CanisterWorm binary target
- `/tmp/.pg_state` — CanisterWorm state tracking file
- `~/.config/sysmon/sysmon.py` — LiteLLM persistence dropper
- `~/.config/systemd/user/sysmon.service` — LiteLLM systemd persistence
- `~/.config/systemd/user/pgmon.service` — CanisterWorm systemd persistence
- `~/.local/share/pgmon/service.py` — CanisterWorm backdoor
- `tpcp.tar.gz` — Exfiltration archive filename (spoofed as PostgreSQL-related)
- `.pth` files (e.g., `litellm_init.pth`) in Python site-packages directories — LiteLLM v1.82.8 persistence that triggers on any Python invocation

Network indicators

- `scan.aquasecurtiy[.]org` (45.148.10.212) — Trivy payload C2 (note typosquatted domain)
- `models.litellm.cloud` — LiteLLM credential exfiltration endpoint
- `checkmarx.zone/raw` (83.142.209.11) — Checkmarx binary payload delivery

- `tdtqy-oyaaa-aaaae-af2dq-cai.raw.icp0.io` — CanisterWorm C2 hosted on Internet Computer blockchain (resistant to traditional takedowns)
- Outbound connections to `*.icp0.io` or `*.ic0.app` — ICP blockchain API endpoints (decentralized C2)
- `169.254.169.254` — IMDS queries from CI/CD runners (the Trivy payload scraped AWS metadata for IAM role credentials)

Kubernetes indicators

- Pods named `node-setup-*` in the `kube-system` namespace — the LiteLLM and Checkmarx payloads deploy privileged pods to every cluster node with host filesystem access

GitHub indicators

- Any repository named `tpcp-docs` or `docs-tpcp` in your organization — these indicate the fallback exfiltration mechanism was used and your credentials were successfully stolen
- Updated tags in GitHub repositories

If you find IoCs in your GitHub environment, you will need to consider that any secrets or credentials are now compromised, as are any systems those credentials grant access to.

Package / File hashes (for verification)

Artifact	SHA-256
litellm 1.82.7 wheel	`8395c3268d5c5dbae1c7c6d4bb3c318c752ba4608cfcd90eb97ffb94a910eac2`
litellm 1.82.8 wheel	`d2a0d5f564628773b6af7b9c11f6b86531a875bd2d186d7081ab62748a800ebb`
Compromised proxy_server.py	`a0d229be8efcb2f9135e2ad55ba275b76ddcfeb55fa4370e0a522a5bdee0120b`
litellm_init.pth	`71e35aef03099cd1f2d6446734273025a163597de93912df321ef118bf135238`

Quick detection commands

```
# Search for .pth persistence files
find site-packages -name "litellm_init.pth"

# Check for persistence dropper
ls -la ~/.config/sysmon/

# Check for CanisterWorm artifacts
ls -la /tmp/pglog /tmp/.pg_state 2>/dev/null

# Check systemd persistence
systemctl --user list-units | grep -E "sysmon|pgmon"

# Check Kubernetes for lateral movement pods
kubectl get pods -n kube-system | grep node-setup

# Search npm lockfiles for known-compromised packages
grep -r "jest-preset-ppf\|babel-plugin-react-pure-component" package-lock.json

# Search for tpcp-docs exfiltration repos in your GitHub org
gh repo list YOUR_ORG --json name -q '.[].name' | grep -iE "tpcp|docs-tpcp"
```

Detailed IOC references

For comprehensive and up-to-date IOC lists, refer to:

- CanisterWorm IOCs
- Aqua / Checkmarx / LiteLLM IOCs
- Telynx IoCs

Phase 2: Remediation

If your investigation reveals exposure — or even potential exposure — immediate remediation is required. The general principle is straightforward but sobering: **treat any environment that ran a compromised component as fully compromised.**

Immediate containment

Remove the malicious components.

Replace affected versions of Trivy, LiteLLM, telynx, CanisterWorm-compromised, npm packages, and any Checkmarx Actions with known-clean versions or remove them entirely. For GitHub Actions, switch to SHA-pinned references pointing to verified clean commits.

Kill persistence mechanisms.

The TeamPCP payloads install multiple persistence mechanisms. Find and remove all of them:

```
# Disable and remove systemd persistence
systemctl --user stop sysmon.service pgmon.service 2>/dev/null
systemctl --user disable sysmon.service pgmon.service 2>/dev/null
rm -f ~/.config/systemd/user/sysmon.service ~/.config/systemd/user/pgmon.service
systemctl --user daemon-reload

# Remove dropper files
rm -rf ~/.config/sysmon/
rm -f /tmp/pglog /tmp/.pg_state

# Remove .pth persistence (LiteLLM v1.82.8)
find /usr -name "litellm_init.pth" -delete 2>/dev/null
find ~/.local -name "litellm_init.pth" -delete 2>/dev/null
```

Clean up Kubernetes lateral movement. If you find `node-setup-*` pods, remove them and audit the nodes they ran on:

```
kubectl delete pods -n kube-system -l app=node-setup
# Then audit affected nodes for further persistence
```

Credential revocation/rotation

This is the most critical and most painful step. As LiteLLM's own guidance puts it bluntly: **"Rotate all the secrets accessible in whatever environment"** ran the affected code.

The TeamPCP credential harvester was comprehensive. It targeted: SSH private keys (50+ filesystem paths); AWS, GCP, and Azure access keys and temporary credentials; CI/CD tokens (GitHub, GitLab, Jenkins); container registry credentials; Kubernetes service account tokens and kubeconfig files; npm authentication tokens; cryptocurrency wallet keys; environment

variables containing secrets; AWS IMDS IAM role credentials; and secrets held in runner process memory.

In practical terms, this means:

- Rotate all cloud provider credentials (AWS access keys, GCP service account keys, Azure client secrets) that were accessible from affected CI/CD runners or developer machines
- Rotate all GitHub PATs and fine-grained tokens. Revoke and regenerate GitHub App installation tokens
- Rotate npm, PyPI, and container registry tokens
- Rotate SSH keys used for repository access or server authentication
- Rotate Kubernetes service account tokens and regenerate kubeconfigs
- Rotate any database credentials, API keys, or third-party service tokens that were present as environment variables or in config files on affected systems
- Invalidate any active sessions that may have been established using stolen credentials

This is a large effort and carries its own risk of causing disruption. Prioritize based on the blast radius of each credential — start with the ones that grant the broadest access (cloud IAM keys, GitHub PATs with org-level scope) and work outward.

A critical lesson: incident rotation ≠ routine rotation

Aqua Security's experience provides the most important remediation lesson from this entire campaign. Their credential rotation failed because it was performed as a routine rotation — issue new credentials, migrate, verify, then revoke old ones — rather than an incident-response rotation.

> **Routine rotation:** Issue new → migrate → verify → revoke old. This is safe when no attacker has access.
>
> **Incident rotation:** Revoke immediately → reissue out-of-band → deploy only after the access path is eliminated.

The difference matters because an attacker with active access can observe the rotation process. In Aqua's case, new credentials were issued before the compromised ones were revoked, and the attacker captured the refreshed tokens during the gap. If you are rotating credentials in response to a suspected compromise, assume the attacker is watching and act

accordingly: revoke first, accept the temporary disruption, and reissue through a channel you are confident the attacker cannot observe.

Component-specific remediation references

- **Trivy** — GitHub Security Advisory GHSA-69fq-xp46-6x23
- **LiteLLM** — Immediate Actions for Affected Users
- **CanisterWorm (npm)** — Endor Labs: CanisterWorm
- **Checkmarx** — Endor Labs: TeamPCP Isn't Done

Phase 3: Prevention

Containing this specific incident is necessary but not sufficient. The TeamPCP campaign has highlighted structural weaknesses in how most organizations manage their CI/CD supply chain — weaknesses that, as Dan Lorenc (CEO of Chainguard) put it bluntly, reflect a platform design problem: *"The design of Actions is plain irresponsible today and ignores a decade of supply chain security work from other ecosystems. What's missing is the will to make secure the default instead of the opt-in."*

Lorenc identifies six architectural failures in GitHub Actions that enabled this campaign: mutable tags without transparency logs; overpermissive default `GITHUB_TOKEN` (pre-2023 repos retain write-all by default); the dangerous `pull_request_target` trigger; unsafe template syntax that drops user-controlled values into shell scripts without escaping; transitive dependency gaps (SHA pinning only covers the outer action, not its dependencies); and no publication controls for Actions (any public repo is an action — no formal publish gate).

Not all of these are within your control to fix. But many of the defensive measures are. This section covers the hardening steps we are working through ourselves and recommend to others, organized by urgency.

GitHub Actions hardening

The industry made significant progress with GitHub Actions hardening after the tj-actions incident, but many organizations — including us — didn't achieve total prevention. You don't have to look deeply into our GitHub posture to see some recent pinning work that was prompted by this incident.

Require SHA pinning

At this point, checking the **"Require actions to be pinned to a full-length commit SHA"** box in your GitHub organization settings (or implementing an equivalent compensating control) should be an immediate goal for every organization. This prevents workflows from referencing mutable tags that can be silently rewritten.

But SHA pinning alone is not enough

Pinning to a SHA addresses the tag-rewriting attack vector, but it doesn't solve everything. You also need to ensure:

- **The overall workflow is secure.** The initial Trivy compromise came through a `pull_request_target` workflow that allowed untrusted code execution with access to repository secrets. SHA pinning wouldn't have prevented this — the problem was in the workflow design, not the Action reference. Scan your workflows for dangerous patterns including untrusted checkouts, injection vulnerabilities, overly broad token permissions, and exposed secrets.

- **The pinned SHAs themselves are clean.** When you pin (or re-pin after an incident like this), you have to verify that the commit you're pinning to is legitimate. TeamPCP pointed all Trivy Action tags to shadow commits — orphan commits not reachable from any branch. A SHA that points to a shadow commit is just as dangerous as a rewritten tag. Verify that your pinned SHAs correspond to commits on a legitimate branch in the upstream repository.
- **SHA updates are carefully reviewed.** Any future change to a pinned SHA — whether through Dependabot, Renovate, or a manual update — represents a moment where a malicious commit could be introduced. Treat SHA updates in GitHub Actions with the same scrutiny you would give a dependency version bump in your application code. Use Dependabot or Renovate to manage SHA-pinned upgrades so you don't fall behind on legitimate updates.
- **Watch for imposter commits.** Imposter commits in a forked repository can bypass GitHub Action security settings and execute malicious code. The GitHub UI displays a warning for imposter commits, but the CLI, API, and Actions do not. This is an active area of risk that SHA pinning alone does not fully address.

Workflow security beyond pinning

Review all workflows for these additional risk factors:

- **`pull_request_target` with untrusted checkout** — The root cause of the Trivy breach. Any workflow using this trigger that checks out PR code must be refactored or removed.
- **Overly broad `GITHUB_TOKEN` permissions** — Set to read-only by default at the organization level.
- **`secrets: inherit` and `toJSON(secrets)`** — Both patterns expose all secrets to every job in a reusable workflow chain.
- **Injection vulnerabilities** — Untrusted expressions (PR titles, branch names, commit messages) interpolated directly into `run:` steps can allow arbitrary code execution.

Dependency management

Dependencies — really, dependency installs — should be gated at every point in your development lifecycle: local development, CI/CD, and runtime.

Gate dependencies with a firewall

A package firewall that intercepts installs and checks packages against malware feeds is critical infrastructure. But it's important to understand the inherent timing gap: malware has to exist before it can be detected and blocked. Malware feeds, including ours, will always have some delay.

This means a robust firewall should not only block known-malicious packages but also **warn or block developers when they are installing a package version that has not yet been analyzed.** A brand-new version that hasn't been evaluated by any feed is a higher risk than one that has been analyzed and cleared.

Codify runtime dependencies

To gate dependencies at runtime, they need to be codified. Dependencies that are installed ad-hoc on production hosts or in running containers — outside the normal build process — represent shadow supply chain risk that no scanner will catch. Ensure that every dependency in your runtime environment traces back to a lockfile that has been scanned and verified.

Scan and verify at every stage

Dependencies in code need to be scanned for known vulnerabilities and malware. But scanning alone is not sufficient — you also need to verify integrity. Lock files should be committed, hash-checked, and monitored for unexpected changes.

Secrets management

LiteLLM's remediation guidance is simple but sobering: "Rotate all the secrets accessible in whatever environment." This is a full-blown compromise, and it is yet another call to action to move off static credentials.

Move to temporary credentials

This is easier said than done, but many providers now offer feature-rich token exchange mechanisms. Moving to temporary, scoped credentials (such as OIDC-based authentication for cloud providers and CI/CD systems) greatly limits exposure once an attacker's foothold is removed. It also eliminates the enormous effort of rotating all static secrets in your environment without breaking things — no small feat.

Eliminate GitHub PATs

GitHub Personal Access Tokens come up repeatedly as a foothold in supply chain attacks. They are often over-privileged, they are tied to individual users rather than organizations (making them hard to administer and audit), and they persist indefinitely unless manually revoked.

Consider alternatives such as:

- **GitHub Apps with installation tokens** — scoped to specific repositories, automatically expiring, and centrally managed
- **OIDC-based token exchange** — for CI/CD workflows that need to authenticate to GitHub, eliminating long-lived tokens entirely
- Chainguard's octo-sts — purpose-built for secure, short-lived GitHub token issuance

Scope secrets narrowly

Even where static credentials remain necessary, apply least-privilege principles aggressively. Every secret should be scoped to the minimum access required and available only to the specific workflows and environments that need it. Organization-wide secrets shared across all repositories are a single point of compromise.

GitHub organization settings

Beyond workflow-level hardening, review your organization-level GitHub settings:

- **Organization Settings → Actions → General → Allow select actions.** Restrict which Actions can run in your organization rather than allowing all public Actions by default.
- **Organization Settings → Repository → General → Immutable releases for all repos.** GitHub Immutable Releases (GA since October 2025) prevent tag rewriting — the exact technique TeamPCP used. This is not enabled by default.
- **Break overprivileged bot accounts into single-purpose tokens.** The `aqua-bot` PAT that was stolen had write access to the entire Trivy ecosystem. A single credential should never grant that breadth of access.

CI/CD network egress monitoring

The TeamPCP attack was actually first detected through anomalous outbound network connections from CI/CD runners. Implement network egress monitoring on your runners:

- **StepSecurity Harden-Runner** — Baselines normal network behavior for your workflows and alerts on anomalies. This is the most directly relevant tool for GitHub Actions environments.
- **Sysdig Falco rules** — Configure rules to detect IMDS access (`169.254.169.254`), `curl`/`wget` exfiltration patterns, and `/proc/mem` reads from CI/CD runner processes.
- **Decentralized C2 patterns** — Monitor for outbound connections to `*.icp0.io`, `*.ic0.app`, and blockchain API endpoints. CanisterWorm's use of Internet Computer Protocol for C2 was a first in documented supply chain attacks, and it represents a new class of infrastructure that traditional domain-based takedowns cannot address.

Runtime security monitoring on CI/CD runners

Deploy runtime detection capabilities that can identify compromise even when preventative controls fail:

- `/proc/*/mem` reads — The credential stealer reads runner process memory to extract secrets. This is highly anomalous behavior in a CI/CD context.

- IMDS credential access — Queries to the AWS Instance Metadata Service from processes that don't normally need it.
- Encrypted file uploads — Large encrypted payloads being sent to external domains during CI/CD runs.
- Systemd service creation — Installation of user-level systemd services during package installation or CI/CD execution.
- Unauthorized npm/PyPI publishes — Monitor your package registry accounts for unexpected version publishes. CanisterWorm's propagation mechanism relies on using stolen tokens to republish packages with injected code.
- Kubernetes anomalies — Privileged pods appearing in `kube-system` with names matching `node-setup-*` are a strong indicator of TeamPCP lateral movement.

Verify security tooling integrity

The central irony of this attack is that a security scanner became the weapon. Going forward, verify the integrity of your security tooling itself:

- cosign + Rekor transparency log for binary verification of security tools before deployment.
- **SLSA provenance attestations** for security tool artifacts — verify that binaries were built from the expected source and build system.
- **Consider locally-cached copies** of critical security tools instead of pulling from public registries on every run. This adds operational overhead but removes the dependency on upstream registry integrity at execution time.

Establish a dependency cool-down policy

Automated dependency management tools like Renovate can be configured to wait for a configurable number of days before upgrading a package. This "cool-down" period allows time for the community to detect and flag malicious versions before they're automatically pulled into your environment. The LiteLLM malicious versions were live for approximately three hours, so a simple cool-down policy of even 24 hours would have avoided exposure.

Getting help from Endor Labs

Alongside this white paper and the individual compromise blogs linked above, Endor Labs is offering a free supply chain security assessment for anyone concerned about the security of their software supply chain.

Appendix: Reference data

MITRE ATT&CK Mapping

The TeamPCP campaign maps to 35 techniques across 11 MITRE ATT&CK tactics. Key technique mappings include:

Technique	Description	TeamPCP Usage
T1195.002	Supply Chain: Software Supply Chain	Tag poisoning (trivy-action, setup-trivy, KICS)
T1195.001	Supply Chain: Software Dependencies	npm worm (CanisterWorm), PyPI (LiteLLM)
T1078.004	Valid Accounts: Cloud Accounts	Stolen service accounts (aqua-bot, ast-phoenix)
T1003.007	OS Credential Dumping: /proc Filesystem	Runner memory scraping via /proc/PID/mem
T1552.001	Unsecured Credentials in Files	50+ filesystem paths for SSH keys, cloud creds
T1552.005	Steal Cloud Instance Metadata	AWS IMDS queries for IAM role credentials
T1543.002	Systemd Service	Persistence via user-level systemd units
T1571	Non-Standard Protocol	ICP blockchain C2 (first documented abuse)
T1537	Transfer Data to Cloud Account	tpcp-docs repos on victim GitHub accounts
T1560.003	Archive via Custom Method	AES-256-CBC + RSA-4096 OAEP encryption

Resources

Endor Labs research

- CanisterWorm: Malicious npm Packages Deploy Self-Propagating Supply Chain Worm
- TeamPCP Isn't Done: LiteLLM, Checkmarx, and the Expanding Blast Radius
- Surprise! Your GitHub Actions Are Dependencies, Too
- GitHub Action Security Policies Documentation
- TeamPCP Strikes Again: Telnyx Compromised Three Days After LiteLLM

Vendor advisories and remediation

- GitHub Security Advisory: GHSA-69fq-xp46-6x23
- LiteLLM Security Update and Immediate Actions

Tools referenced in this guide

- Chainguard octo-sts — Short-lived GitHub token issuance
- Sigstore cosign + Rekor — Binary verification and transparency logs

CVE reference

CVE-2026-33634 — CVSS 9.4 (Critical) — CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:C/C:H/I:H/A:H