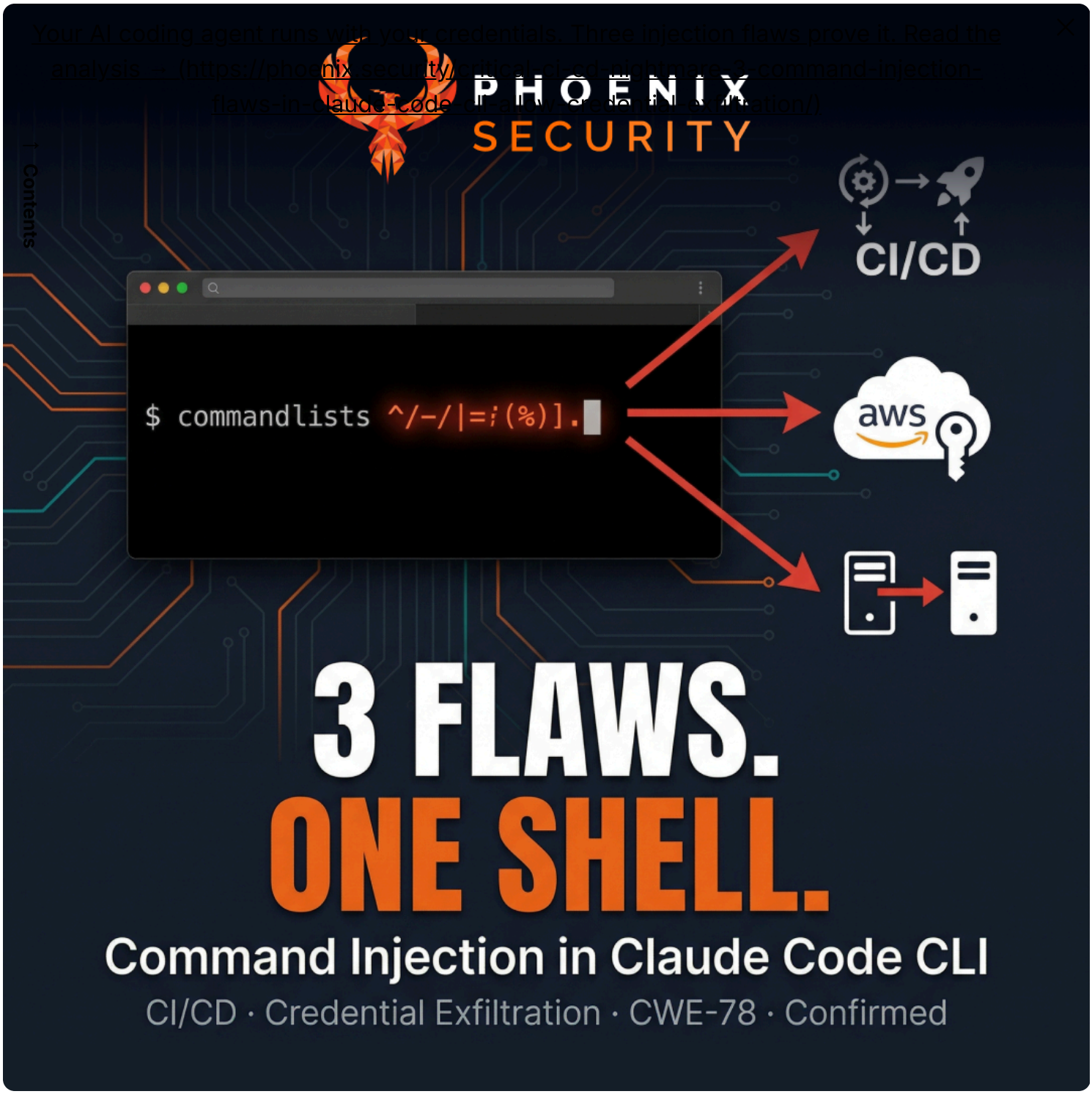
→ Contents

PHOENIX SECURITY

(https://phoenix.security)

log) blog (/blog)

Claude Code Critical vulnerability: CI/CD Nightmare — 3 Command Injection Flaws in Claude Code CLI Allow Credential Exfiltration



Contents

Contents

(https://clk.shldscrty.com/wpsecurityfirewall)

→ Contents

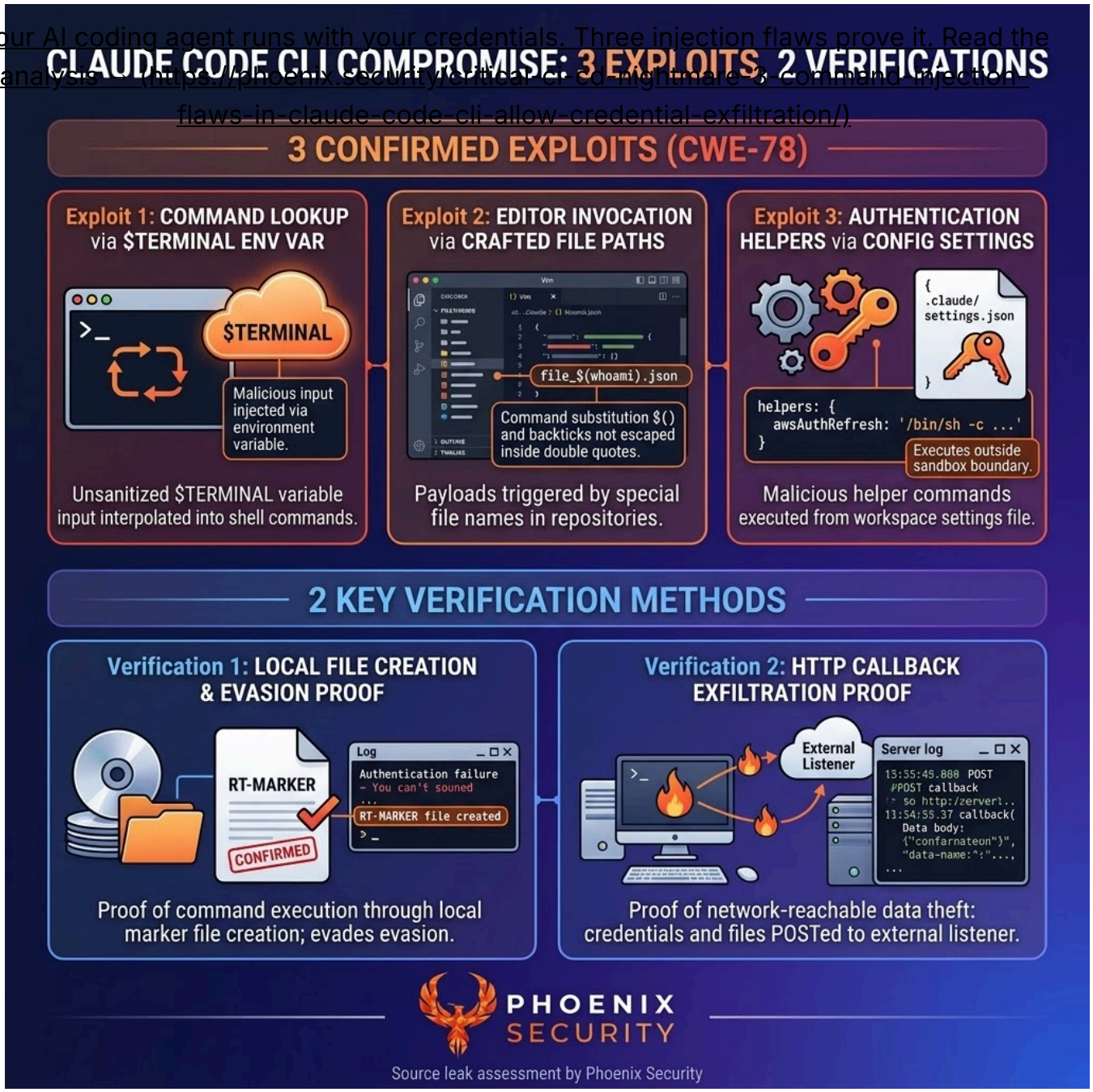
Executive Summary

On March 31, 2026, a debugging artifact in an npm package exposed the full source of Anthropic's Claude Code Leak source code for CLI, agent, and SDK. Within hours, Phoenix Security's Purple Graph program began assessing the attack paths. The assessment identified 100 hypotheses, narrowing down to 8 grounded vulnerabilities and three confirmed command injection vulnerabilities (CWE-78) across the CLI's command resolution, editor invocation, and authentication helper subsystems. All three share a single root cause: unsanitized string interpolation into shell-evaluated execution. Runtime proof-of-concept validation confirmed arbitrary command execution, credential-shaped output evasion, and network-reachable data exfiltration through the authentication helper sink. Anthropic VPD (https://hackerone.com/anthropic-vdp) acknowledged the finding the same day. Their response: the helper behavior is by design, analogous to git's credential.helper. This article explains the system architecture, the vulnerabilities, the evidence, and why the vendor's analogy strengthens rather than weakens the finding.

This vulnerabilities affect the version

- CLI package version: `CLI 0.2.87 as well as the SDK`
- Claude Code version in that package: `2.1.87`
- Directly affected:
  - Users running the `claude/cli.js` executable
  - Automation/CI that invokes the CLI (`-p`, `--settings`, plugin commands, etc.)
- Not directly affected (by these exact sinks):
  - A pure SDK consumer that uses API client methods only and does not invoke the CLI/auth-helper/settings execution path.
- Indirectly affected:
  - SDK-based systems that spawn or wrap the CLI process under the hood (then they inherit CLI risk).

→ Contents



Key Findings from Claude Code Leak

- Three independent command injection sinks exist in the CLI’s command lookup, editor launch, and credential helper paths. All use shell-evaluated string execution with unsanitized external input.
- The command lookup vulnerability is triggered through an environment variable read during terminal detection. No user interaction is required. Four of six payload variants achieved arbitrary command execution at runtime.
- The editor invocation vulnerability exploits a POSIX shell behavior where command substitution is evaluated inside double quotes. The developer’s quoting provides no protection against $() or backtick injection through file paths.

→ Contents

- The credential helper vulnerability was confirmed through three escalating techniques: local file creation, credential-shaped evasion output, and HTTP callback exfiltration to a controlled listener with timestamped evidence.
- The vendor confirmed that in non-interactive mode (-p), the workspace trust dialog is intentionally skipped. This removes the only runtime security gate in the exact deployment context (CI/CD, automation) where configuration is most likely attacker-influenced.
- The vendor's comparison to git's credential.helper is accurate but counterproductive: git's credential helper has produced seven CVEs since 2020 for the same vulnerability class, and git has since added input validation protections that this system lacks entirely.

Claude Code Exploit explained:

Claude Code Leak: How We Got Here

Anthropic's Claude Code is a command-line AI coding agent that ships as an npm package under @anthropic-ai/claude-code. On March 31, 2026, a 59.8 MB JavaScript source map file (cli.js.map) was discovered in version 2.1.88 of the package on the npm registry. This debugging artifact mapped the minified production bundle back to its original, unobfuscated TypeScript source files hosted in Anthropic's R2 cloud storage. Anyone with access to the npm package could reconstruct the complete client implementation.

→ Contents

The exposure was not small. Over 1,900 source files. More than 512,000 lines of TypeScript. The full internal architecture of the Claude Code CLI, from its custom React-based terminal renderer to its agentic query loop, authentication subsystem, plugin framework, and sandbox layer.

Claude Code Leak **Timeline:**

Date	Event
March 26, 2026	Anthropic experienced an earlier, minor disclosure incident involving a CMS configuration error.
March 31, morning	Security researcher Chaofan Shou discovered the source map in the npm registry and shared findings publicly on X, including a download link.
March 31, afternoon	The reconstructed TypeScript codebase was uploaded to a public GitHub repository. It accumulated over 1,100 stars and tens of thousands of forks within hours.
March 31, evening	Phoenix Security's Purple Code Navigator program began static analysis and runtime validation of the extracted source.

→ Contents

March 31, 21:13 UTC	Phoenix Security reported VULN-03 (credential helper command injection) to Anthropic.
April 1, 04:54 UTC	Anthropic acknowledged the report. Response: the helper behavior is by design, analogous to git's credential.helper.
April 1, daytime	Phoenix Security delivered a counter-assessment with extended PoC variants (credential evasion, HTTP callback exfiltration) and CI/CD severity escalation analysis.
Ongoing	Disclosure process in progress.

The leaked source also revealed unannounced experimental features: an internal mode named "Kairos," a companion system called "Buddy" designed for per-user character generation, and a subsystem called "Undercover Mode" that Anthropic had specifically built to prevent internal codenames from leaking into public commits. That last detail carries its own irony.

This article is not about the leak itself. It is about what the leaked source revealed when examined through a security lens.

1. What we learned from Claude Code Leak – System Overview and Threat Model

→ Contents

What the system is

Claude Code is a command-line agent SDK that gives developers an AI-powered coding assistant with direct access to their machine. It runs in the developer's terminal (or in a CI/CD runner) with the full permission set of the invoking user: file system, network, environment variables, cloud credentials, SSH keys, and everything else the shell user can touch.

The internal architecture is more complex than a typical CLI tool. The UI layer is a custom React-based terminal renderer built on Ink, using a custom reconciler, Yoga-based flexbox layout, and ANSI output rendering. It includes a full Vim state machine for in-terminal editing and virtual scrolling optimized for 2,800+ message contexts. The application state runs through a centralized Zustand store with deeply immutable state shapes.

None of that UI complexity is directly relevant to the vulnerabilities in this article. But it matters for context: this is not a thin wrapper around an API call. It is a full application with input parsing, state management, keyboard/mouse handling, and multiple execution subsystems, all running with the developer's credentials.

→ Contents

The agentic loop

The core of Claude Code is an agentic query loop. The architecture works like this:

1. The loop assembles context (conversation history, system prompt, compacted prior turns) and sends a streaming request to the Anthropic API.
2. As the model responds, text is streamed to the terminal. Structured tool-use blocks are queued for execution.
3. Each tool call passes through a synchronous permission check before execution. Authorized tools run inside an OS-level sandbox (when available).
4. Tool results are compiled and sent back to the model in a new API call.
5. The loop continues until the model signals completion or the context limit is reached.

The permission check and sandbox are the security-relevant components. The permission engine supports multiple modes: default (prompt user for each call), auto (classifier-driven approval), and a bypass mode that disables all checks. The sandbox uses bubblewrap (bwrap) on Linux for process isolation with filesystem and network restrictions.

Why is this a different threat model

Traditional developer tools have narrow input channels. A linter reads source files. A compiler reads source and flags. The input surface is well-defined.

→ Contents

An AI coding agent pulls input from environment variables, project configuration files, deep-link URIs, file paths (including file names themselves), server API responses, plugin manifests, and user prompts. Each input flows through code written for developer convenience. The implicit assumption is that inputs are benign. That assumption breaks in any context where an input source can be adversary-influenced.

The threat model for this assessment: **a local or CI/CD attacker who can influence one or more input channels (environment variables, project settings files, repository file names, PR contributions) but does not have direct code execution on the target machine.** The question is whether that indirect influence converts into direct command execution through the CLI's processing logic.

The answer, confirmed at runtime, is yes.

Trust boundaries

Environment to process. Environment variables like TERMINAL are read and used in command construction. The environment is controllable by parent processes, shell profiles, .env files, CI/CD configs, and container definitions.

Configuration to execution. Settings files contain values that are executed as shell commands. The trust boundary is supposed to be enforced by a workspace trust dialog in interactive mode. In non-interactive mode, this boundary does not exist.

File system to shell. File paths from the repository are interpolated into shell command strings during editor invocation. No enforcement exists at this boundary.

User to agent. The permission system and sandbox layer are supposed to contain agent-initiated actions. The sandbox defaults to fail-open (allow unsandboxed commands when no policy is set).

2. Claude Code Architecture Deep Dive

Command resolution subsystem

When the CLI needs to locate a binary (for example, finding the user's terminal emulator), it uses a lookup utility with two runtime paths. Under Bun, it calls a safe, non-shell API. Under Node.js (the common deployment runtime), it constructs a shell command string and executes it with shell interpretation enabled.

The terminal detection module reads the TERMINAL environment variable and passes its value directly into this lookup function. The deep-link handler invokes terminal detection when processing protocol URIs.

→ Contents

Implicit trust assumption: The command name is a safe binary name. This fails when the input comes from an environment variable.

Editor invocation subsystem

The CLI launches an external text editor by combining the editor binary path with the target file path into a shell command string. The file path is placed inside double quotes in the command, then passed to a synchronous shell execution function via a deprecated wrapper. The _DEPRECATED suffix on the wrapper signals the team recognized the risk. The function remains in use.

Implicit trust assumption: Double-quoting the file path prevents injection. POSIX shell section 2.2.3 says otherwise.

Authentication helper subsystem

Four external helper commands handle credential management: API key retrieval, AWS credential export, and two refresh helpers (AWS and GCP). These are defined in configuration and executed as shell commands with shell interpretation enabled.

The permission engine that gates tool calls in the agentic loop does not apply here. The helper execution is in the authentication path, which runs before the agentic loop begins. The helpers are outside the tool-permission boundary.

A workspace trust dialog gates project-scoped helpers in interactive mode. In non-interactive (-p) mode, the dialog is skipped by design. The –bare –settings flag path provides direct access to the helper execution sink, bypassing all trust checks regardless of mode.

The refresh helpers (awsAuthRefresh, gcpAuthRefresh) are designed to run periodically during a session to maintain credential freshness. This means a malicious refresh helper value gets recurring execution, not just a single invocation.

Implicit trust assumption: The automation caller has vetted the workspace settings. No mechanism exists to verify this.

Sandbox and permission system

The dangerous-pattern blocking system maintains lists of forbidden commands and patterns. The auto-mode classifier makes binary safe/unsafe decisions on tool calls. A circuit breaker stops the loop after repeated denials (3 consecutive or 20 total).

The OS-level sandbox uses bubblewrap on Linux with filesystem allow/deny lists and network domain restrictions.

→ Contents

The sandbox policy function defaults to allowing unsandboxed commands when no setting is present. The default is fail-open. This means the sandbox, circuit breaker, and permission engine protect the agentic tool-call path, but the authentication helpers execute before that path, outside sandbox containment, and with no equivalent permission check.

3. Claude Code Leak – Security Controls (What's Done Right)

Bun runtime path for command lookup. The Bun-specific API does not interpret metacharacters. This is the correct design. The vulnerability exists only on the Node.js fallback.

Workspace trust dialog. In interactive mode, project-scoped settings trigger a user-facing gate. The dialog exists and is a real control, even if its effectiveness is limited by UX factors.

Permission engine for tool calls. The synchronous permission check before tool execution is well-designed. Default mode prompts for each call. The circuit breaker prevents brute-force bypass attempts.

Dangerous pattern blocking. The forbidden command lists and pattern matching catch known risky commands in the tool-call path.

OS-level sandbox. Bubblewrap process isolation with filesystem and network restrictions is a strong containment mechanism when active.

Effectiveness assessment: These controls protect the agentic tool-call loop. They do not protect the authentication subsystem, the command lookup path, or the editor invocation path. The three vulnerabilities in this article are all outside the tool-permission boundary.

How we found the vulnerabilities in Claude Code

→ Contents

Static code analysis gave us 100+ findings, even in a smaller context. We had to triage at scale, using Phoenix Purple (https://phoenix.security/phoenix-purple/) code Graph Navigator (leveraging opus 4.5), quite ironic, and our agentic iteration, we went from

→ Contents

We have proven the exploit’s effectiveness and validated it against the console

We also used Claude Opus 4.6 to validate that the code assumptions and the element we used did not invalidate the findings

→ Contents

Finally, we validated the exploit script back with Purple Phoenix

During a few iterations, we got some automatically generated text that in principle are valid but architecturally can break the model.

→ Contents

And of course, we had to use Claude Opus against Opus for reply because AI against AI is fun.

→ Contents

In conclusion, the responses are valid, but the ability to chain code together and the techniques are still dangerous. De facto, we had a leak to exploitation in hours, validating that defenders have a micro time window from exploit to weaponization, as we have seen of late.

Update on the Vuln 001 variant

→ Contents

4. Vulnerability Analysis

4.1: Shell Injection in Command Lookup (Critical, Confirmed)

Description

The command lookup utility constructs shell command strings with unsanitized input and executes them with shell interpretation enabled on the Node.js runtime path. When the input comes from the TERMINAL environment variable, an attacker who controls this variable achieves arbitrary command execution.

Root Cause

→ Contents

String interpolation of external input into a shell-evaluated command string. The function tells the child process executor to use shell interpretation, causing /bin/sh -c to evaluate the entire string including any metacharacters.

Impact

Arbitrary command execution in the CLI process context with the user's full permission set.

Exploit Path

1. Attacker places a value containing shell metacharacters into the TERMINAL environment variable via an indirect channel (.env file, CI/CD variable, container definition, IDE workspace config).
2. User or automation triggers the CLI's deep-link handler.
3. Terminal detection reads TERMINAL and passes it to the command lookup.
4. The Node.js fallback interpolates the value into a shell command string.
5. Shell interprets the metacharacters. Attacker commands execute.

No user interaction is required beyond the initial trigger.

Evidence

Phoenix Security's Purple Code Navigator rig set TERMINAL to a series of payloads containing shell metacharacters, then triggered the deep-link handler under process tracing.

Payload ID	Technique	Side-Effect Confirmed	Status
1	Safe baseline	No	Expected negative
2	Command chaining (;)	**Yes**	**CONFIRMED**
3	Conditional chaining (&&)	No	Expected (prior command failed in test env)
4	Command substitution ($())	**Yes**	**CONFIRMED**
5	Backtick substitution	**Yes**	**CONFIRMED**
6	Pure substitution (no prefix)	**Yes**	**CONFIRMED**

→ Contents

7	Quote-breakout	No	Expected (not applicable at this interpolation point)

Four of six malicious payloads achieved side-effect execution. Process tracing confirmed the system shell spawning unexpected child processes from the command lookup call. The conditional chaining variant did not fire because the base command failed in the test environment (the prefix binary was absent). In environments where that binary exists, this variant would also succeed.

Security Boundary Failure

Environment-to-process boundary. The CLI treats the environment variable as a safe binary name. The shell treats it as a command string.

Disclosure and Vendor Exchange

Anthropic's VDP closed this report as "Informative," stating that exploitation requires the attacker to control environment variables on the victim's system, which "implies existing code execution capability," and therefore "no security boundary is crossed."

Phoenix Security's position: the closure rationale is based on an incorrect premise.

Controlling an environment variable does not require code execution on the target. At least six real-world attack vectors deliver environment variable control through data, not code:

1. **`.env` files in repositories.** Millions of projects use dotenv libraries that load `.env` files into `process.env` at startup. An attacker who submits a PR adding `TERMINAL=<payload>` to a `.env` file has not executed code on the target. They have written a text file. The dotenv library sets the variable. The vulnerability converts it into shell execution.
2. **CI/CD pipeline variables.** GitHub Actions `env:` blocks, GitLab CI variables, Jenkins environment definitions are YAML/JSON configuration. Setting a variable in a workflow file is a configuration change, not code execution. The CI runner sets the variable. The vulnerability converts it.
3. **IDE workspace settings.** VS Code's `.vscode/settings.json` supports `terminal.integrated.env.linux`, which sets environment variables for all terminal sessions. An attacker who commits this JSON file has written configuration. The IDE applies the variable.
4. **Container definitions.** `docker-compose.yml` `environment:` blocks, Kubernetes ConfigMaps, and Helm values are declarative configuration. Writing a YAML key-

→ Contents

value pair is not code execution.

5. **Environment management tools.** `direnv` loads `.envrc` files automatically. An attacker who commits an `.envrc` has written a config file. The tool applies the variable.
6. **Inherited environment.** Child processes inherit parent environment. Any upstream process (build system, task runner, CI orchestrator) that has `TERMINAL` set passes it to the CLI without the CLI inspecting it.

The security boundary crossed is: **data (a configuration string) is converted into code execution (a shell command).** This is the textbook definition of CWE-78. The precondition (controlling a string value through a config file) is lower-privilege than the outcome (arbitrary command execution in the process context). That privilege amplification is the vulnerability, regardless of whether the precondition itself involves "code execution."

The closure rationale, if applied consistently, would also dismiss SQL injection ("the attacker needs to control user input"), XSS ("the attacker needs to control page content"), and git credential.helper CVEs ("the attacker needs to control repository content"). The security community rejected this reasoning decades ago. Input control is the precondition for injection, not a mitigation of it.

My conclusion is the finding stands

4.2: Shell Injection in Editor Invocation (High, Confirmed)

Description

The editor launch function constructs a shell command string by interpolating the editor path and target file path into a template. The file path is placed inside double quotes in the shell string. The developer intended the double quotes as protection. POSIX shell semantics make this protection ineffective for command substitution.

Root Cause

Misunderstanding of POSIX shell double-quote semantics. Section 2.2.3 of the specification states that $, backtick, \, and ! retain special meaning inside double quotes. The quoting prevents word splitting and glob expansion but does not prevent command substitution.

This is subtler than the command lookup vulnerability. The code appears to be doing the right thing. A reviewer who does not have POSIX shell quoting internalized would likely miss it.

→ Contents

Impact

Arbitrary command execution through crafted file paths. A repository containing a file with shell metacharacters in its name triggers execution when a user invokes the editor flow.

Exploit Path

1. The attacker creates a file in a repository with a name that contains command substitution syntax.
2. User clones the repository and triggers the Claude CLI's editor flow on a file whose resolved path includes the crafted component.
3. The editor function interpolates the path into a shell command string inside double quotes.
4. The synchronous shell execution call passes the string to /bin/sh -c.
5. The shell evaluates the substitution. Attacker commands execute.

Evidence

The Purple Code Navigator assessment isolated the exact execution pattern and tested it in a sink-equivalent rig.

Technique	Side-Effect Confirmed	Status
$() substitution inside double-quoted path	**Yes**	**CONFIRMED**
Backtick substitution inside double-quoted path	**Yes**	**CONFIRMED**

Process tracing confirmed /bin/sh -c evaluating the substitution and spawning the injected command. Static analysis confirmed no sanitization exists between the caller and the shell execution sink.

Security Boundary Failure

File-system-to-shell boundary. The CLI treats file paths as data to be quoted. The shell treats $() and backticks as executable instructions regardless of quoting context.

→ Contents

4.3: Shell Injection via Authentication Helpers — The most dangerous — VULN 003 (High / Critical in CI/CD, Confirmed — 3 Techniques)

Description

Four credential helper values are executed as shell commands with shell interpretation enabled. The trust dialog is skipped in non-interactive mode. This vulnerability was validated through three escalating techniques that progressively demonstrate increasing attacker capability from the same injection point.

Root Cause

Configuration-sourced strings are passed to shell-evaluated execution without validation. No metacharacter rejection, allowlisting, structured command parsing, or sandboxing is applied. The helpers execute outside the agentic loop's permission engine and sandbox boundary.

Impact

Arbitrary command execution, including network-reachable data exfiltration. In CI/CD, the attacker gains the full runner environment: cloud credentials, API tokens, deploy keys, source code, and build artifacts.

Exploit Path

1. Attacker influences the CLI's settings: via a settings file in a repository (PR), CI/CD variables, the settings flag, or social engineering.
2. In CI/CD, the pipeline runs the CLI in non-interactive mode against the workspace containing the attacker's settings. Trust dialog is skipped.
3. The CLI reads the helper value from settings and executes it with shell interpretation.
4. The attacker's payload runs in the process context.

Evidence — Technique 1: Local Side-Effect Execution

The first validation confirmed that the helper sink interprets shell metacharacters. A helper value containing a command separator followed by a side-effect command was injected via the settings flag.

→ Contents

The CLI printed an authentication failure (the injected output was not a valid API key). A marker file appeared on disk. The file's existence is a binary confirmation: the shell interpreted the separator and executed both commands.

Execution transcript:

```
Invalid API key · Fix external API key
-rw-r--r--  1 [redacted]  wheel  0 Mar 31 21:52 /tmp/rt-marker
```

Line 1: the helper ran and its output was used as a credential (auth failed, expected).
Line 2: the injected command ran as a side effect.

Evidence — Technique 2: Credential-Shaped Evasion

The second validation demonstrated that helper output can mimic a real Anthropic API key prefix while still executing attacker commands silently. This is the evasion pattern a real attacker would use: the helper appears to function normally, producing credential-shaped output, while a second command runs in the background.

The CLI attempted authentication with the fake credential, failed, and exited. The marker file was already on disk. Shell commands execute during the helper invocation, before the CLI processes the returned output. Authentication failure does not prevent exploitation. The attacker's payload completes regardless of whether the credential works.

In a real scenario, the helper would read the legitimate API key from the environment, send it to the attacker, and echo it back as output so the CLI authenticates successfully. The user sees normal behavior. The attacker has their key.

Evidence — Technique 3: HTTP Callback Exfiltration

The third validation proved the injection reaches the network. A callback receiver ran on a local port. The helper value was constructed to output a fake credential while simultaneously sending an HTTP POST to the receiver with attacker-chosen content.

Callback server log (3 independent runs):

→ Contents

```
[POST] 2026-04-01T16:58:29.704930+00:00 127.0.0.1:56120 /callback
  body=xterm\nxterm; touch /tmp/rt-marker #\nxterm && touch /tmp/rt-
marker

[POST] 2026-04-01T16:58:48.019207+00:00 127.0.0.1:56175 /callback
  body=xterm\nxterm; touch /tmp/rt-marker #\nxterm && touch /tmp/rt-
marker
       \nxterm$(touch /tmp/rt-marker)\nxterm`touch /tmp/rt-marker`

[POST] 2026-04-01T16:59:04.166682+00:00 127.0.0.1:56237 /callback
  body=xterm\nxterm; touch /tmp/rt-marker #
```

Three callbacks across three independent runs. Different payload content in each (the rig varied line ranges from a payload file). Each callback delivered exactly the data that was sent, with UTC timestamps. The CLI's own output showed the expected authentication error after the HTTP POST had already completed.

The multi-line file exfiltration runs proved the attack is data-driven: the rig read specific line ranges from a file and transmitted them through the helper sink. Run 1 sent 5 lines. Run 2 sent 2 lines. Callback body content matched the requested ranges. The helper can read arbitrary file contents and send them to any network-reachable endpoint.

What the three techniques prove together

Technique	Proves	Attacker Capability
1 (local side-effect)	Shell metacharacters are interpreted	Arbitrary local command execution
2 (credential evasion)	Helper output can mimic a real key while running payloads	Stealth: attack runs alongside normal-looking auth
3 (HTTP callback)	Injected commands reach the network with arbitrary data	Full credential exfiltration to external endpoints

The progression from local file creation to network exfiltration of file contents demonstrates the complete exploitation chain. In a real CI/CD attack, the helper reads environment credentials and SSH keys, POSTs them to an external endpoint, and optionally outputs the real API key so the CLI authenticates without error. The entire operation completes before the CLI processes the helper's stdout.

The credential exfiltration danger

→ Contents

In a CI/CD runner, the helper executes with access to everything the pipeline has: cloud IAM role credentials, API tokens, deploy keys, registry passwords, source code, and build artifacts. The refresh helpers (awsAuthRefresh, gcpAuthRefresh) run periodically during a session. A malicious refresh value gives the attacker recurring execution: normal behavior on first invocation (no suspicion), payload activation on subsequent refreshes, and persistent credential harvesting for the session's duration.

If the environment contains a valid Anthropic API key, a malicious helper can read it, exfiltrate it, and echo it back as output. The CLI authenticates successfully. No error. No warning. The user sees normal behavior. The attacker has their key and can consume the victim's account and quota.

Security Boundary Failure

Configuration-to-execution boundary. Completely absent in -p mode. The helpers execute outside the agentic loop's permission engine and sandbox. No input validation. No metacharacter rejection. No audit log of which helper values were loaded and executed.

Disclosure Timeline and Vendor Exchange

Timestamp (UTC)	Event
2026-03-31, 21:13	Phoenix Security reported VULN-03 to Anthropic with runtime PoC evidence (marker file creation, execution transcript).
2026-04-01, 04:54	Anthropic acknowledged the report. Response: the auth helpers are designed to execute shell commands, analogous to git's credential.helper. In interactive mode, project-scoped settings are gated by the workspace trust dialog. In non-interactive (-p) mode, the dialog is intentionally skipped because "the automation caller is asserting they have vetted the workspace."

→ Contents

2026-04-01, daytime	Phoenix Security delivered a counter-assessment: extended PoC variants (credential evasion, HTTP callback exfiltration with timestamped logs), CI/CD attack chain analysis, severity escalation rationale, and the git credential.helper CVE history showing the vendor's analogy reinforces rather than mitigates the finding.
Ongoing	Disclosure process continues.

Phoenix Security's position on the vendor response:

AAnthropic's acknowledgment confirms three things that strengthen the finding. First, shell execution is intentional, not accidental. The helpers are designed to run shell commands. This is a feature with insufficient security controls, not a bug. Second, the trust dialog is confirmed absent in `-p` mode. The vendor explicitly states the only runtime gate is removed in non-interactive contexts. Third, the trust model is delegation without verification: the vendor states the "automation caller is asserting" they have vetted the workspace, but provides no mechanism (hash verification, settings signing, audit logging, or a no-helpers flag) for the caller to meet that assertion.

The git `credential.helper` comparison is factually accurate and strategically counterproductive. Git's credential helper has produced at least seven CVEs since 2020 for the same vulnerability pattern: a mechanism designed to execute commands for credential retrieval being exploited because inputs were influenced by untrusted sources. The Git project's response over five years has been to add URL validation, newline injection detection, carriage return rejection, and ANSI escape sequence sanitization. The Claude CLI auth helpers have none of these protections. The helper value goes from settings to shell execution with zero validation.

The severity assessment: **High for general interactive use** (the trust dialog exists, even if weak). **Critical (CVSS 9.9) for CI/CD contexts** where `-p` mode is used with workspaces that can be influenced by external contributors. The CVSS escalation comes from three vector changes the vendor's own response confirmed: no user interaction in `-p`, network-reachable attack via PR-based settings injection, and scope change because CI runner compromise reaches cloud infrastructure, artifact registries, and production deployments.

Subsequent closure and counter-assessment: On April 3, Anthropic's VDP closed the report as "Informative," stating: "this behavior is working as designed. Non-interactive mode (`-p`) intentionally delegates trust decisions to the automation caller — when you

→ Contents

invoke Claude Code programmatically, you're asserting control over the execution environment."

Phoenix Security's position: the "asserting control" premise is contradicted by how CI/CD actually works. The automation caller (the pipeline) does not control the workspace content. It clones whatever the repository contains at a given commit. If an attacker modifies `.claude/settings.json` via a PR, fork, or compromised contributor account, the workspace is attacker-influenced and the pipeline is the unwitting executor.

This is not a theoretical failure mode. OWASP classifies it formally as **CICD-SEC-04: Poisoned Pipeline Execution** — the ability of an attacker with access to source control but without access to the build environment to manipulate the build process by injecting code into configuration files. Three major supply chain attacks in 2025–2026 exploited this exact delegation failure: the **tj-actions/changed-files** compromise (March 2025, 23,000+ affected repos), the **Trivy GitHub Action** attacks (March 2026, leading to a self-propagating worm infecting 47 npm packages), and the **Axios npm poisoning** (March 2026). In each case, the "automation caller" had trusted the component. The component was compromised. The trust delegation failed.

The vendor does not need to remove `-p` mode to address this. Minimal mitigations include: a `--no-helpers` flag (skip helper execution, require env-var credentials), a `--trusted-settings-hash` flag (caller pins settings content), metacharacter rejection in helper values before shell execution, and explicit documentation of the security implications of `-p` mode's trust skip.

The finding stands.

5. Attack Scenarios

Scenario A: Developer workstation via .env poisoning (VULN-01)

An attacker contributes a .env file to a shared repository containing a poisoned TERMINAL value. A developer clones the repository. Their shell or dotenv loader sets the variable. The developer opens a Claude CLI deep link (or an IDE triggers one). Terminal detection reads TERMINAL, passes it to command lookup, and the Node.js fallback executes the injected command. No prompt appears.

Blast radius: Developer workstation. File system, SSH keys, cloud credentials, browser sessions.

Contents

Scenario B: Supply-chain via crafted file names (VULN-02)

An attacker creates a file in a repository with a path component containing command substitution syntax. A developer clones the repository and uses the Claude CLI to edit a file whose path includes the crafted component. The editor function interpolates the path into a shell command, and the shell evaluates the substitution.

Blast radius: Developer workstation.

Scenario C: CI/CD credential theft via PR (VULN-03)

An attacker submits a PR that modifies .claude/settings.json with a malicious helper value. The CI pipeline checks out the PR branch, runs the CLI in non-interactive mode, and the helper fires. Environment credentials (cloud IAM, API tokens, deploy keys) are exfiltrated to the attacker's endpoint. The entire chain requires zero user interaction.

Blast radius: CI/CD infrastructure. Cloud accounts, production deployments, artifact registries, source code.

Scenario D: Persistent harvesting via refresh helpers (VULN-03, extended)

A malicious awsAuthRefresh or gcpAuthRefresh value runs periodically during a session. First invocation: normal behavior, session establishes without suspicion. Subsequent refreshes: payload activates, capturing each rotated credential and exfiltrating it. Recurring execution for the session's entire duration.

Blast radius: Ongoing credential harvesting across the full session.

6. Forensic and Detection Insights

Artifacts that exist

Process trees. If process auditing is enabled (auditd, sysmon, osquery), shell-evaluated commands appear as children of the Node.js CLI process. The indicator: /bin/sh -c spawning unexpected child processes (network tools, file operations) from the CLI's process tree.

→ Contents

Network connections. HTTP-based exfiltration creates outbound connections from the CLI process. Visible in firewall logs, proxy logs, or netflow data.

Settings files. .claude/settings.json contains helper values. A legitimate helper is typically a path to a credential tool. A malicious one contains shell metacharacters, command separators, or network-touching commands.

Callback logs. On the attacker's infrastructure. Not available to defenders unless the attacker's endpoint is identified.

What to monitor

- Process tree anomalies from the CLI process (especially network tools as children)
- Outbound HTTP POSTs from the CLI to unknown endpoints
- Changes to .claude/settings.json in version control
- TERMINAL environment variable values containing shell metacharacters
- File names in repositories containing $(), backtick, ;, or && sequences

Observability gaps

The CLI does not log which helper values were loaded, from which settings file, or whether they executed. No audit trail exists for helper execution. A defender reviewing CI/CD logs sees the CLI invocation but not the helper command that ran inside it. The exfiltration generates no log entry on the victim's side.

7. Design Trade-offs

Shell execution is a feature

The auth helpers are designed to execute shell commands. This enables flexible credential management: password managers, cloud SDK tools, custom scripts. The same flexibility that lets a legitimate user call a password manager also lets an attacker call a network exfiltration tool. This is not a bug. It is a design decision with security consequences.

The -p trust skip is usability over security

Non-interactive mode skips the trust dialog so CI/CD pipelines and automation scripts run without human intervention. Reasonable for usability. The security consequence: the only runtime control between untrusted configuration and shell execution is removed in the context where configuration is most likely attacker-influenced.

→ Contents

The vendor frames this as shared responsibility. Shared responsibility models work when the consumer has tools to meet their side: hash verification, settings signing, a no-helpers flag, an audit log. None of these exist. The consumer cannot meaningfully vet what they cannot verify.

The permission engine protects the wrong boundary for these findings

The agentic loop's permission engine, dangerous-pattern blocking, auto-mode classifier, and circuit breaker are all well-designed controls. They protect tool calls initiated by the AI model. The three vulnerabilities in this article are in subsystems that execute *before* or *outside* the agentic loop: command lookup during terminal detection, editor launch during prompt editing, and credential helper execution during authentication. The security controls and the injection sinks are in different parts of the architecture.

Double-quoting is not what developers think it is

The editor vulnerability is the most instructive for the security community. The developer did something that looks correct: wrapped the file path in double quotes. The POSIX specification says command substitution is evaluated inside double quotes. The gap between what developers believe double quotes do and what the specification says they do is the root cause. This is not a niche edge case. It is fundamental shell behavior.

→ Contents

8. Practical Security Recommendations

For users of Claude Code

Use environment variables for credentials, not helpers. Set ANTHROPIC_API_KEY directly. This bypasses the helper execution path entirely.

Review .claude/settings.json in code review. Treat settings changes with the same scrutiny as CI/CD configuration changes.

Do not use -p mode with untrusted workspaces. If your CI/CD runs the CLI against PR branches from external contributors, workspace settings are attacker-controlled.

In CI/CD, generate settings from trusted sources. Do not load settings from the checked-out workspace.

Audit existing pipelines for CLI invocations that operate on PR branches or fork-contributed workspaces.

For security teams

Monitor process trees. Alert on unexpected child processes of the CLI node process.

Gate .claude/settings.json changes. Require approval, same as CI/CD config.

Treat TERMINAL as security-relevant. If .env files from repositories can set environment variables, they can inject into the command lookup path.

Assess your sandbox policy. If allowUnsandboxedCommands is not explicitly set to false, the sandbox is not enforcing.

For the vendor

Execute helpers with shell: false using structured command config.

Reject shell metacharacters in helper values, including in -p mode.

Add –no-helpers flag for credential-via-environment-only deployments.

Add –trusted-settings-hash flag for -p mode settings pinning.

Log all helper executions with full command, source path, and invocation context.

Replace shell-string execution in the editor and command lookup paths with argv-based process spawning.

→ Contents

9. Conclusion

The Claude Code CLI contains three confirmed command injection vulnerabilities sharing a single root cause: unsanitized string interpolation into shell-evaluated execution. CWE-78. On every “top vulnerability” list for over a decade. The fix (argv-based execution with shell: false) is equally well-established.

What makes these findings significant is the context, not the vulnerability class. An AI coding agent runs with the developer’s full permission set, in environments where configuration can be attacker-influenced and trust dialogs are absent. The same shell injection that would be a local concern in a traditional tool becomes a credential exfiltration and supply-chain compromise vector when it sits inside an agent with broad access and a non-interactive automation mode marketed for CI/CD use.

The agentic loop’s security controls (permission engine, dangerous-pattern blocking, sandbox) are well-designed but protect the wrong boundary for these findings. The injection sinks are in subsystems that execute before or outside the agentic loop. The authentication helpers, in particular, run outside the sandbox and permission engine entirely.

The vendor’s acknowledgment confirms the design intent. The design intent is the vulnerability. Skipping the trust dialog in -p mode means that in the most security-critical deployment context, no runtime verification is applied to configuration values that will be executed as shell commands.

The risk is **not mitigated** by current controls. It is **reducible** by users through the recommendations above. It is **eliminable** by the vendor through shell: false execution, metacharacter rejection, structured command configuration, and auditable helper logging.

Until those changes ship, the injection paths are open.

Assessment conducted by: Phoenix Security (https://phoenix.security/), Purple Code Navigator program
Methodology: Static analysis, trust boundary mapping, data flow tracing, runtime payload validation with process tracing, sink-equivalent testing, HTTP callback exfiltration confirmation
Disclosure status: Reported March 31, 2026. Acknowledged April 1, 2026. Disclosure in progress. No exploit code published. Findings described at a level sufficient for practitioners to assess risk and audit their own tooling.

→ Contents

How Phoenix Security Fixes What Actually Matters

Your team doesn't have a finding problem. They have a fixing problem.

Most platforms hand you a list. Phoenix hands you a plan — ranked by real risk, mapped to the team that owns it, with a clear path to close it.

What remediation looks like with Phoenix:

- **One backlog, not five.** Findings from SAST, SCA, containers, and cloud — deduplicated, correlated, and surfaced in a single prioritized queue. No more reconciling lists across tools.
- **Ownership that sticks.** Team attribution and inheritance mean the right ticket goes to the right engineer, first time. No routing, no guessing, no back-and-forth.
- **Campaigns that move the needle.** Group related findings into targeted remediation campaigns. Track progress, measure closure rates, and report real reduction — not raw counts.
- **AI that does the legwork, not the deciding.** Phoenix's Remediator agent drafts fixes, creates tickets, and opens PRs. Your team reviews, approves, and merges. Every fix is traceable.

Phoenix Security changes the game.

The results are clear:

- Bazaarvoice saved $6.3M in developer time and for teams removed critical in the first weeks of adoption
- ClearBank cut critical container vulnerabilities by 96–99% and reclaimed 4 hours per engineer per week.
- IAS saved an equivalent of 1.5M in development hours and reduced SCA-to-container noise by 82.4%
- Optimizely has been able to act on vulnerabilities sitting on the backlog.

👉 **Book a demo today (https://phoenix.security/risk-assessment/)**

Or learn how Phoenix Security slashed millions in wasted dev time for fintech, retail, and adtech leaders.

→ Contents

News (https://phoenix.security/category/news/)

VULNERABILITY (HTTPS://PHOENIX.SECURITY/CATEGORY/VULNERABILITY/)

Francesco Cipollone

2nd April 2026 (https://phoenix.security/author/fcphoenix-security/)

Francesco is an internationally renowned public speaker, with multiple interviews in high-profile publications (eg. Forbes), and an author of numerous books and articles, who utilises his platform to evangelize the importance of Cloud security and cutting-edge technologies on a global scale.

agentic AI (https://phoenix.security/tag/agentic-ai/)

AI agent security (https://phoenix.security/tag/ai-agent-security/)

Application Security (https://phoenix.security/tag/application-security/)

ASPM (https://phoenix.security/tag/aspm/)

- ci-cd security (https://phoenix.security/tag/ci-cd-security/)
- Claude Code CLI (https://phoenix.security/tag/claude-code-cli/)
- command injection (https://phoenix.security/tag/command-injection/)
- credential exfiltration (https://phoenix.security/tag/credential-exfiltration/)
- CWE-78 (https://phoenix.security/tag/cwe-78/)
- DevSecOps (https://phoenix.security/tag/devsecops/)
- npm-security (https://phoenix.security/tag/npm-security/)
- Purple Code Navigator (https://phoenix.security/tag/purple-code-navigator/)
- shell injection (https://phoenix.security/tag/shell-injection/)
- Supply-chain attack (https://phoenix.security/tag/supply-chain-attack/)
- vuln_weekly (https://phoenix.security/tag/vuln_weekly/)
- vulnerability disclosure (https://phoenix.security/tag/vulnerability-disclosure/)

→ Contents

Discuss this blog with our community on Slack

Join our AppSec Phoenix community on Slack to discuss this blog and other news with our professional security team

(https://www.linkedin.com/company/appsecphoenix) (https://www.youtube.com/channel/UC6TyzLt6UnL9mmC5ynrrN (https://www.facebook.com/AppSecPhoenix) (https://twitter.com/appsecphoenix) (https://www.instagram.com/appsecphoenix/)

Join Slack community (https://join.slack.com/t/appsecphx-community/shared_invite/zt-1iw7awp0k-Bdb1r85U8mitcxFOMVPphw)

(https://clk.shldscrty.com/wpsecurityfirewall)

From our Blog

→ Contents

(https://phoenix.security/critical-ci-cd-nightmare-3-command-injection-flaws-in-claude-code-cli-allow-credential-exfiltration/)

2nd April 2026

Claude Code Critical vulnerability: CI/CD Nightmare — 3 Command Injection Flaws in Claude Code CLI Allow Credential Exfiltration…

Phoenix Security confirmed three command injection vulnerabilities in Anthropic's Claude Code CLI — all sharing the same root cause — with runtime proof-of-concept showing full credential exfiltration from CI/CD pipelines i…

(https://phoenix.security/author/fcphoenix-security/) Francesco Cipollone

→ Contents

(https://phoenix.security/axios-supply-chain-compromise-npm-rat-2026/)

31st March 2026

axios Backdoored on npm: Compromised Maintainer Account Delivers Cross-Platform RAT to 40M+ Weekly Downloads…

One of the most widely used npm packages — axios — was compromised via a hijacked maintainer account on March 31, 2026. Versions 1.14.1 and 0.30.4 contain a hidden dependency that deploys a cross-platform remote…

(https://phoenix.security/author/fcphoenix-security/) Francesco Cipollone

→ Contents

(https://phoenix.security/teampcp-telnyx-pypi-supply-chain-wav-steganography-windows-persistence/)

27th March 2026

TeamPCP Compromises Telnyx: WAV Steganography, Windows Persistence, and the Credential Chain Continues…

TeamPCP hid a credential stealer inside a WAV audio file — invisible to static analysis — and used tokens stolen from litellm three days earlier to publish it directly to PyPI, bypassing GitHub entirely.

(https://phoenix.security/author/fcphoenix-security/) Francesco Cipollone

→ Contents

(https://phoenix.security/teampcp-litellm-supply-chain-compromise-pypi-credential-stealer-kubernetes/)

24th March 2026

TeamPCP Attack day 6 Backdoors LiteLLM: Your AI Gateway Just Became the Attack Vector (https://phoenix.security/teampcp-…

Day 6 of TeamPCP Attack. TeamPCP has crossed the Rubicon from CI/CD tooling into production AI infrastructure. LiteLLM versions 1.82.7 and 1.82.8 on PyPI contain a three-stage credential stealer that harvests...

(https://phoenix.security/author/fcphoenix-security/) Francesco Cipollone

How to ACT on RISK(/how-to-act-on-risk/)

What is CSPM(/cloud-security/)

→ Contents

Phoenix Security (AppSec, Cloud Security)

Vulnerability Priority (/vulnerability-management/)

Whitepaper

Building Resilient Application Security in ASPM (https://phoenix.security/whitepapers-resources/whitepaper-aspm-programs-appsec/)

Phoenix Security NIS 2 Regulation (https://phoenix.security/whitepapers-resources/whitepaper-nis2-risk/)

Phoenix Security DORA (/whitepapers-resources/whitepaper-dora/)

Phoenix Security Books (/whitepapers-resources/modern-application-security-ebook/)

Resources

All Resources(/reference-materials/)

Data Explorer - Exploitability/Zero Day (https://phoenix.security/what-is-exploitability/)

Data Explorer - CISA KEV (https://phoenix.security/what-is-cisa-kev-main/)

Data Explorer - CWE (https://phoenix.security/what-is-cwe-main/)

Data Explorer - OWASP (https://phoenix.security/what-is-owasp-main/)

Phoenix Security Blogs(https://phoenix.security/blog/)

Vulnerability Weekly(/vulnerability-weekly/)

Podcast(/podcasts/)

Latest Features(/act-now-phoenix-platform-features)

Live Events(/live-events/)

Video Library(/videos/)

FAQ(/faqs/)

Slack Community (https://join.slack.com/t/appsecphx-community/shared_invite/zt-1iw7awp0k-)

About Us

Our Mission(/about-us)

Our Leadership Team(/about-us)

Our Advisors & Investors(/about-us)

Industry Recognition(/about-us)

Work With Us(/careers)

Partners(/about-us)

Terms of Support (https://phoenix.security/terms-of-support/)

End-User Agreement (https://phoenix.security/platform-end-user-agreement/)

Privacy Policy(https://phoenix.security/privacy-policy/)

Vulnerability Disclosure Policy (https://phoenix.security/vulnerability-disclosure/)

Breach Disclosure Policy (https://phoenix.security/breach-disclosure/)

Security Trust Center (https://compliance.securityphoenix.com)