AVERLON

SEVEN ATTACKS, ONE PATTERN: WHY THE DEVELOPMENT ENVIRONMENT IS NOW IN THE CROSSHAIRS

Between October 2025 and April 2026, seven attacks hit six layers of the software development environment. Mapping the pattern — and what it means for the enterprise threat model.

July 2026

OVERVIEW

Between October 2025 and April 2026, seven separate attacks hit six different parts of the software development environment. Different groups. Different techniques. Different targets. Some were financially motivated. Some were state-sponsored. Some were opportunistic.

What they had in common was where they struck: not production systems, not customer databases, not the assets security programs are traditionally organized to protect. They hit the development environment. The part of the stack where code is written, built, tested, and shipped. And increasingly, the part where the most sensitive context about how systems work now lives.

This piece maps those seven attacks to the six layers they targeted, and what the pattern says about how the threat model needs to evolve.

1. What the Development Environment Actually Is

Most people picture a developer at a laptop writing code. That is far from the full truth.

The development environment practitioners actually work in every day is a collection of interconnected layers, each with its own trust model, its own credential store, and its own external connections.

This is not a new taxonomy. It maps closely to what the Supply Chain Levels for Software Artifacts framework (SLSA) describes as the software supply chain, and to the build and deployment topology the Cloud Native Computing Foundation (CNCF) has documented for cloud-native systems.

The one layer neither framework fully accounts for yet is the AI toolchain, which has become central to how software is written and has no established governance model in most organizations.

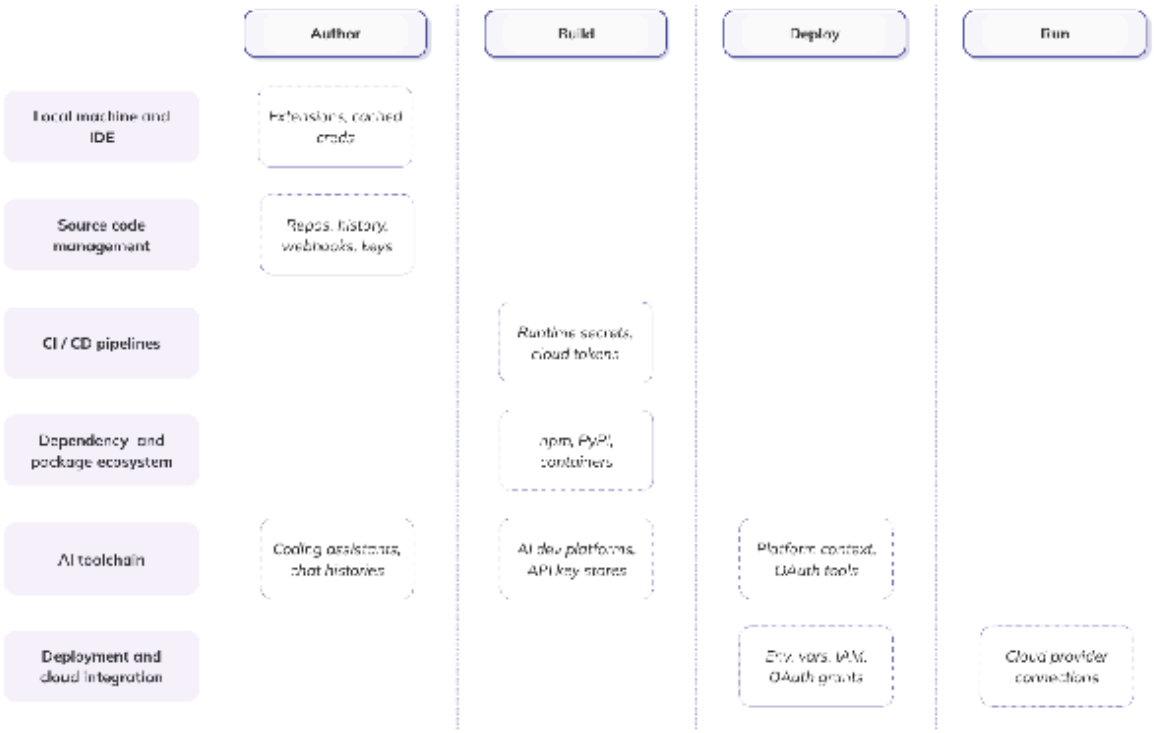

Figure 1: *Six layers. Four phases. Each a distinct target.*

Here is what each layer holds and where it sits in the delivery lifecycle.

The local machine and IDE. This is where code is authored. It holds IDE extensions and plugins, locally cached credentials like SSH keys and cloud CLI tokens stored in files like `~/.aws/credentials`, git configuration, and browser-stored tokens for development platforms. Extensions carry OS-level access by default: they can read files, execute processes, and make network requests. EDR tools monitor developer machines for suspicious process execution and file system changes, but they rarely trace what a trusted extension does with a cloud token it reads through normal file access.

Source code management (SCM). Where code is collaborated on and its complete history maintained. Git, GitHub, GitLab, and Bitbucket all live here. The SCM layer holds repositories and their full commit history, branch protection rules that determine which contributors can merge to protected branches and what review requirements must be met, webhook integrations that fire HTTP requests to external systems whenever repository events occur, and signing keys used to verify commit authenticity. According to GitGuardian's 2025 State of Secrets Sprawl report, 23.8 million hardcoded secrets were added to public GitHub repositories in 2024 alone, a 25 percent increase year over year, and 70 percent of secrets leaked in 2022 remain active today. Secrets committed accidentally don't expire when the commit is deleted. They persist in forks, clones, and mirrors potentially outside of the organization's control.

CI/CD pipelines. Where code is built, tested, and prepared for deployment. GitHub Actions, Jenkins, CircleCI, and their equivalents sit here. The CI/CD layer is the most privileged in the development environment because the build process needs to perform privileged actions: pushing a container image to a registry requires authentication, deploying to a cloud environment requires cloud provider credentials, signing a release artifact requires a signing key. Rather than hardcode those

credentials, modern pipelines retrieve them at runtime from a secrets manager, typically HashiCorp Vault, AWS Secrets Manager, or the platform's own encrypted secrets store, and inject them as environment variables for the duration of the build. That means every pipeline run has live, production-grade credentials in memory, accessible to every tool that executes as part of the build. It also runs code that the organization did not write, at every stage of the build.

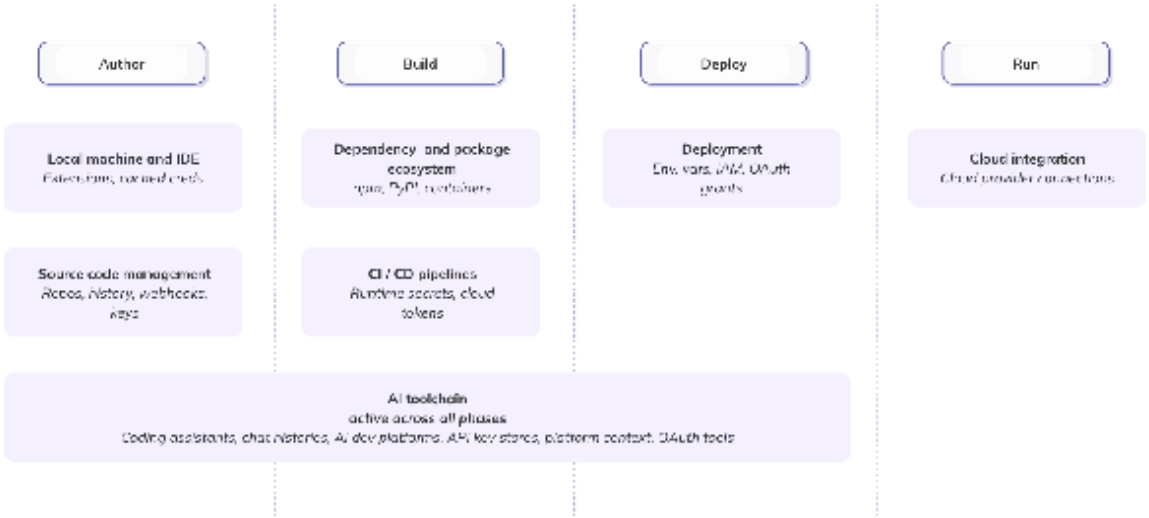

Figure 2: *The development environment spans the entire delivery lifecycle, not just where code is written. The AI toolchain extends across phases.*

Dependency and package ecosystem. Also active at build time. When a build runs, it resolves dependencies from package registries: npm for JavaScript and Node.js, PyPI for Python, container registries for base images. Those dependencies are pulled automatically and bundled into the artifact that ships. According to the 2025 Open Source Security and Risk Analysis report, the average application today includes more than 900 open-source components, the majority of which were never individually reviewed by the team shipping the software. A compromised package in the registry at the moment of build becomes part of the artifact automatically, which is why the LiteLLM packages caused downstream exposure during a window of less than six hours.

The AI toolchain. This layer spans authoring, build, and parts of the deployment stage, and it is the one neither SLSA nor CNCF's documentation meaningfully addresses. It includes AI coding assistants like GitHub Copilot and Cursor, AI-assisted development platforms like Lovable, Bolt, and Replit, and AI gateways like LiteLLM that aggregate API keys for multiple LLM providers into a single integration for use during development and testing. What distinguishes this layer is what accumulates in it over time. Developers use AI coding tools by providing context: they paste error logs, describe architectural decisions, share database schema definitions, ask questions about deployment configurations. Those conversations are stored by the platform. As of 2024, GitHub Copilot had more than 20 million active users and ChatGPT was used regularly by 82 percent of developers surveyed on GitHub. The AI toolchain is now central to how software gets written. The chat histories it accumulates are a compressed record of how systems are built, pasted in one

prompt at a time, sitting in a platform-managed data store that most security programs have never inventoried.

Deployment and cloud integration. Where software is deployed and where the development environment's trust relationships reach production directly. Deployment platforms like Vercel, Railway, and AWS Amplify store environment variables that configure deployed applications: database connection strings, third-party API keys, feature flags, and service credentials. IAM roles grant the deployment process access to cloud resources. OAuth grants connect developer tooling to cloud accounts and SaaS platforms. This is the layer where a credential compromise in a development context becomes a production incident.

Each of those layers was targeted at least once between October 2025 and April 2026. Here is what happened in each one.

2. Seven Attacks Across the Development Environment

The seven attacks between October 2025 and April 2026 were executed by different groups with different techniques and different objectives. Mapped against the layer model, they cover the terrain almost completely.

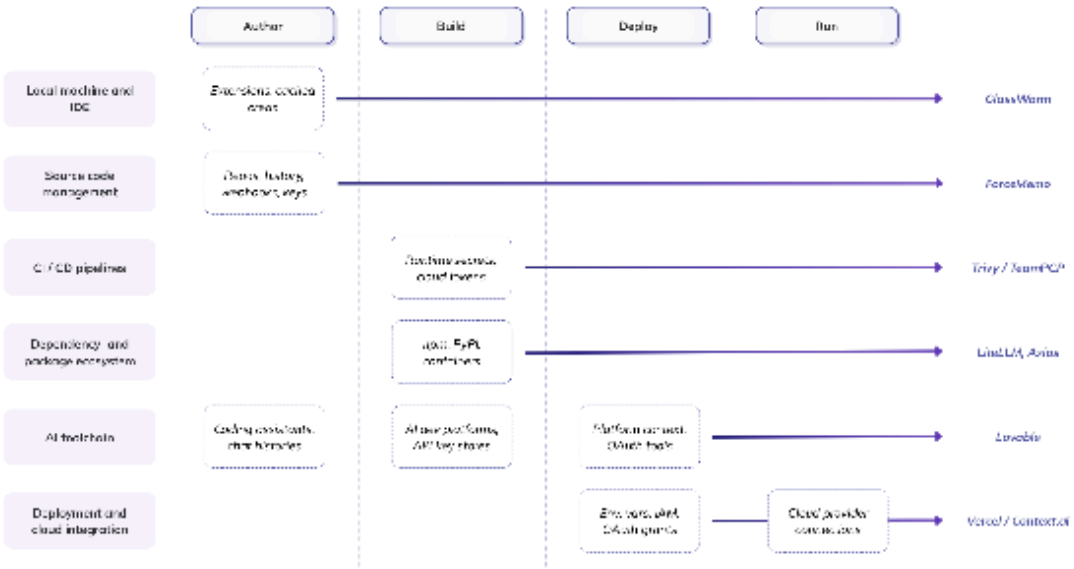

Figure 3: *Seven attacks. Six layers. October 2025 to April 2026.*

Incident 1 — local machine and IDE: GlassWorm.

GlassWorm earned its name from two properties: its code was invisible to human reviewers, hidden using Unicode variation selectors that render as blank space in every code editor, and it was self-propagating, using stolen developer credentials to automatically compromise additional extensions

and repositories without further attacker involvement. It was first documented by Koi Security in October 2025 and reached its largest wave in March 2026.

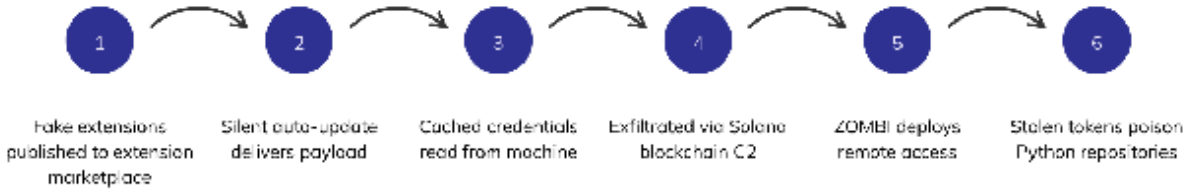

Figure 4: *How one compromised extension became an ecosystem-wide intrusion.*

The attack chain:

1. The attacker published fake extensions to the OpenVSX marketplace, impersonating popular tools: linters, formatters, AI coding assistant integrations, under typosquatted publisher names. Some variants began as clean extensions, later updated to pull in malicious dependencies automatically, so an extension that passed review at install time became a delivery vehicle in a subsequent update.
2. Developers installed them. VS Code extensions auto-update silently. Developers who had installed earlier clean versions received the malicious payload without any prompt or warning.
3. Once active, the extension read locally cached credentials from the developer's machine: npm tokens, GitHub personal access tokens, git credentials, SSH keys, cloud CLI tokens, cryptocurrency wallets, and browser-stored authentication tokens.
4. Credentials were exfiltrated to a C2 server. The primary channel encoded payload URLs in Solana blockchain transaction memos, making the infrastructure resistant to takedown. Google Calendar served as a fallback, with Base64-encoded URLs hidden in event titles. The malware skipped execution on systems with a Russian locale, a standard indicator of Eastern European or state-adjacent threat actor tradecraft.
5. A second-stage payload, ZOMBI, deployed a SOCKS proxy and remote access capability, turning the compromised developer machine into attacker-controlled infrastructure.
6. Stolen GitHub tokens were then used in the ForceMemo follow-on campaign to force-push malware into Python repositories, rewriting git history while preserving the original commit message, author, and date, leaving no pull request or commit trail visible in GitHub's UI.

By mid-March 2026, the campaign had compromised 433 components: 72 VS Code extensions, 88 npm packages, and more than 151 repositories, with over 9 million installs of malicious extensions reported.

The commits carrying the payloads were indistinguishable from routine maintenance. The accompanying changes were realistic: documentation tweaks, version bumps, minor refactors, each stylistically consistent with the target project's codebase. At the scale of 151 repositories, that consistency points to AI-generated cover commits. The same capability available to defenders, applied to the problem of not getting caught.

The blast radius extended far beyond the initial developer machine. Stolen credentials compromised hundreds of GitHub accounts and turned developer machines into proxy nodes for further criminal activity. A single compromised extension propagated through an entire ecosystem without the attacker needing to touch each target individually.

Incident 2 — source code management: ForceMemo.

ForceMemo was the second stage of GlassWorm, executed using GitHub tokens stolen from compromised developer machines. However, what made it a distinct attack is where it operated: not on the developer's local machine, but inside source code repositories themselves.

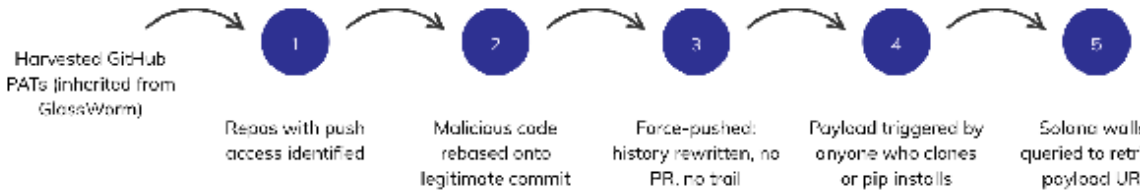

Figure 5: *How stolen credentials became silent injections across 400 Python repositories.*

The attack chain:

1. GlassWorm-harvested GitHub personal access tokens (PATs) gave attackers write access to hundreds of developer accounts
2. Beginning March 8, 2026, attackers identified Python repositories where the compromised accounts had push access: Django apps, machine learning codebases, PyPI packages, and Streamlit dashboards
3. For each target repository, attackers rebased the latest legitimate commit on the default branch with malicious code appended to files like `setup.py`, `main.py`, and `app.py`
4. The changes were force-pushed to the default branch, rewriting git history while preserving the original commit message, author, and author date. No pull request was opened. No review was triggered. No commit trail appeared in GitHub's UI
5. Anyone who ran `pip install` from a compromised repository, or cloned and executed the code, triggered the malware automatically
6. The injected code queried the same Solana blockchain wallet used by GlassWorm to retrieve the payload URL, confirming operational continuity between the two campaigns

The SCM layer's specific vulnerability here was not a misconfiguration or a software flaw. It was the implicit trust model of git itself. A commit from a legitimate author on a legitimate branch passes every automated check. The malicious commits were indistinguishable from real ones because they were made by real accounts, with real credentials, preserving real commit metadata. The only detectable signal was the committer date, which differed from the author date, a discrepancy most pipelines do not monitor.

The blast radius reached more than 400 Python repositories across hundreds of GitHub accounts. Any downstream project that installed from a compromised repository during the exposure window deployed GlassWorm's payload without any connection to the original VS Code extension campaign.

Incident 3 — CI/CD pipelines: Trivy and TeamPCP.

On March 19, 2026, TeamPCP compromised Trivy, Aqua Security's open-source vulnerability scanner embedded in thousands of CI/CD pipelines. The attack poisoned GitHub Actions workflows and Docker images simultaneously, injecting malicious code into official distribution channels. The attack was assigned CVE-2026-33634.

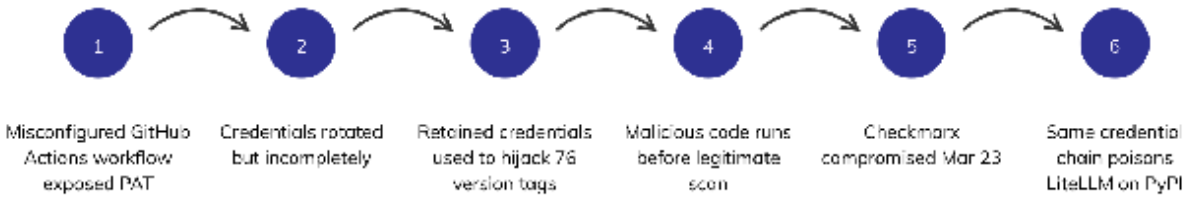

Figure 6: *How an incomplete credential rotation became a six-day cascade.*

The attack chain:

1. In late February 2026, TeamPCP exploited a misconfigured GitHub Actions workflow in the Trivy repository, extracting a privileged Personal Access Token and establishing a foothold in repository automation.
2. Aqua Security detected the intrusion, rotated credentials, and communicated the incident. The rotation was incomplete: not all credentials were revoked simultaneously, and the attacker retained access through the survivors during the multi-day rotation window.
3. On March 19, using the retained credentials, TeamPCP force-pushed 76 of 77 version tags in `aquasecurity/trivy-action` and all 7 tags in `aquasecurity/setup-trivy` to point at malicious commits, simultaneously publishing a malicious Trivy binary designated `v0.69.4`
4. The malicious code ran inside `entrypoint.sh` before the legitimate Trivy scan executed, so pipelines appeared to succeed while secrets were being exfiltrated: API tokens, cloud credentials, SSH keys, Kubernetes tokens, Docker configuration files, and git credentials.
5. Using credentials stolen from the Trivy intrusion, TeamPCP compromised Checkmarx's GitHub Actions for two repositories on March 23.
6. On March 24, malicious versions of LiteLLM were published to PyPI using the same stolen credential chain.

The target selection was deliberate. Trivy runs as a GitHub Action on every PR, every merge, every deployment. It runs with access to pipeline secrets by design. Compromising a security tool that

operates across every pipeline in an organization's build process is not just credential theft. It is gaining access to everything those pipelines touch, at the moment they touch it.

The most actionable technical detail in the entire campaign: the initial intrusion was detected and the standard playbook was followed. It wasn't enough. The rotation was treated as binary: contained or not contained. The actual problem was a graph: which credentials were rotated, what did the unrotated credentials touch, what trust relationships extended from those credentials into downstream systems. The playbook did not ask those questions completely. The attacker retained the access that answered them.

The blast radius extended through every pipeline that ran Trivy during the exposure window, then cascaded through retained credentials to Checkmarx and LiteLLM. Mandiant noted the stolen data would likely be leveraged for months, with the blast radius continuing to expand well after the initial incident was closed.

Incident 4 — dependency and package ecosystem: Axios.

On March 31, 2026, North Korean state-sponsored actors attributed by Google Threat Intelligence Group to UNC1069 compromised Axios, the most widely used JavaScript HTTP client, present in roughly 80 percent of cloud and code environments with over 100 million weekly downloads. The entry point was not a vulnerability in the code. It was the maintainer's trust.

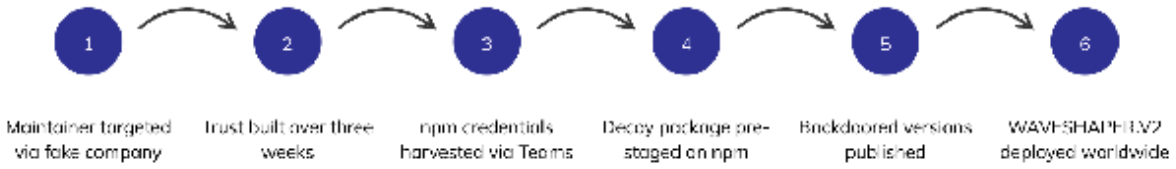

Figure 7: *How three weeks of social engineering poisoned 100 million weekly downloads.*

The attack chain:

1. Beginning around March 12, attackers impersonated the founder of a legitimate company, building a convincing Slack workspace with plausible channel activity, team profiles, and linked social content to establish credibility.
2. After several weeks of exchanges, the lead maintainer was drawn into a Microsoft Teams meeting staged by the attackers to appear as a legitimate business call with multiple participants. The meeting yielded his npm credentials.
3. With credentials in hand, the attacker pre-staged a clean decoy package, `plain-crypto-js`, on npm 18 hours before the attack, giving it a brief legitimate history to evade new-package detection rules.
4. On March 31 at 00:21 UTC, two backdoored versions of Axios, `1.14.1` and `0.30.4`, were published via the hijacked account. Each injected `plain-crypto-js` as a phantom dependency that executed automatically on install with no user interaction required.

5. The payload, WAVESHAPER.V2, deployed a cross-platform backdoor across Windows, macOS, and Linux, establishing persistent remote access and exfiltrating credentials.
6. The malicious versions were live for approximately three hours before npm removed them. Any environment that ran `npm install` during that window deployed the malware automatically, including CI/CD pipelines with auto-updating dependencies.

A mirrored package, `@depup/axios`, republished the compromised version just 17 minutes after its original release, suggesting automated systems extended the blast radius before the original was pulled.

The detection signal was available in retrospect. Authentic Axios releases carry OIDC provenance metadata and SLSA build attestations linking the package back to a specific GitHub Actions run. The malicious versions had neither. That absence is detectable. Most pipelines were not checking for it.

The blast radius reached every environment that resolved to the compromised versions during the three-hour window: developer workstations, CI/CD pipelines, and production applications worldwide. Google's threat intelligence team noted the full breadth remained unclear given the package's ubiquity. Any system that executed the payload should be treated as fully compromised: credentials rotated, systems rebuilt from clean snapshots.

Attack	Layer	What was exposed	Blast radius
GlassWorm	Local machine and IDE	SSH keys, cloud tokens, cryptocurrency wallets	Any service the developer authenticated to from the compromised machine
ForceMemo	Source code management	GitHub repository access, Python codebase integrity	Hundreds of Python repositories across hundreds of GitHub accounts, any downstream project installing from compromised repos
Trivy / TeamPCP	CI/CD pipelines	Pipeline secrets, cloud access tokens, Personal Access Token	Cascaded to Checkmarx and LiteLLM via retained credentials from incomplete remediation
Axios	Dependency and package ecosystem	Maintainer credentials harvested via social engineering, backdoored npm package deployed	Any project with auto-updating axios dependencies that ran npm install during the three-hour window
LiteLLM	Dependency and package ecosystem	API keys for OpenAI, Anthropic, and other LLM providers	Any environment running LiteLLM with unpinned dependencies during the six-hour exposure window
Lovable	AI toolchain	Source code, database credentials, full AI chat history	Any project created before November 2025, exposed for 48 days after first report
Vercel / Context.ai	Deployment and cloud integration	Environment variables, API keys, npm tokens, GitHub tokens	Vercel customers with no direct relationship to Context.ai, reached via single OAuth trust chain

Table 1: *What each attack took, and how far it reached*

Incident 5 — dependency and package ecosystem: LiteLLM.

LiteLLM is an AI gateway library: a universal interface for accessing multiple LLM providers from a single integration. Environments running LiteLLM store API keys for OpenAI, Anthropic, and other providers in one place by design. That aggregation is the product's value proposition. It is also exactly what made it a high-value follow-on target after Trivy.

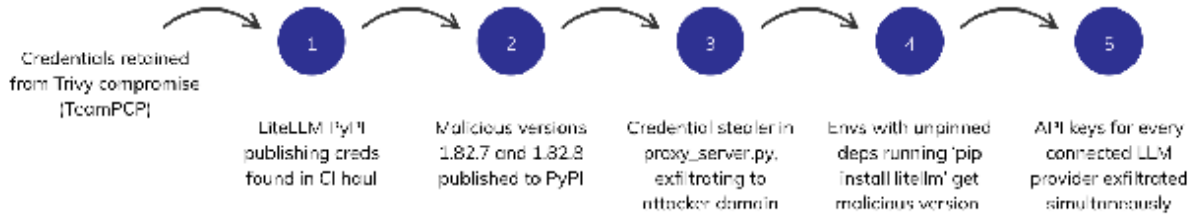

Figure 8: *How one incomplete remediation gave attackers access to every LLM provider credential.*

The attack chain:

1. Using credentials retained from the incomplete Trivy remediation, TeamPCP found LiteLLM's PyPI publishing credentials in the CI haul and on March 24 published malicious versions `1.82.7` and `1.82.8` directly to the registry.
2. The packages contained a credential stealer embedded in `proxy_server.py`, configured to exfiltrate data to `models.litellm[.]cloud`, an attacker-controlled domain designed to look like a legitimate LiteLLM endpoint.
3. Any developer or downstream project with unpinned dependencies that ran `pip install litellm` during the exposure window received the malicious version automatically, with no indication anything had changed.
4. The malicious versions were live for less than six hours before being quarantined. Wiz noted LiteLLM was present in 36 percent of cloud environments they monitored at the time of the incident, signifying the potential for widespread impact.

LiteLLM was not chosen at random. TeamPCP found LiteLLM's PyPI publishing credentials in the Trivy CI haul and recognized what they had access to: a library that aggregates API keys for every LLM provider an organization has connected in a single integration. One compromised installation exposed credentials for OpenAI, Anthropic, and every other provider simultaneously. The six-hour exposure window is also worth naming precisely: automated build systems across thousands of environments can pull, bundle, and ship a compromised package to production in less time than most incident response processes take to mobilize. The exposure window closed. The artifacts it produced did not.

Incident 6 — AI toolchain: Lovable.

A Broken Object Level Authorization (BOLA) vulnerability in the AI-assisted development platform Lovable exposed something categorically different from the other incidents in this pattern: not

credentials, but context.

The vulnerability was in the `/projects/{id}/*` API endpoints, which verified Firebase authentication tokens but skipped ownership checks entirely. Any authenticated free-tier account could reach any project. The flaw was first reported via HackerOne on March 3, 2026. It was closed as a duplicate without escalation. It was publicly disclosed on April 20, 2026, after a researcher demonstrated it remained exploitable. The window between first report and public disclosure: 48 days.

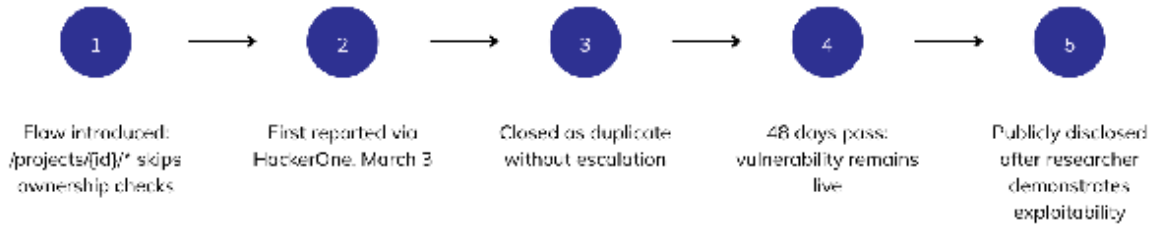

Figure 9: *A 48-day window for attackers to access context.*

What was exposed: source code, database credentials, and the full AI conversation history tied to each project. Every prompt a developer had sent, including pasted error logs, architecture questions, database table definitions, deployment configurations, and credentials shared mid-session. Projects created before November 2025 were affected.

The conversation history is a significant detail. A developer using an AI coding assistant does not think of those conversations as a sensitive data store. They think of them as a productivity tool. But the accumulation of prompts over months of active development is a detailed record of how the system was built: what decisions were made, what the schema looks like, what the error patterns are, what third-party services it connects to. An attacker with that history does not need to probe the system to understand it. They already do.

Apps built on Lovable frequently embed credentials for third-party services including Supabase, Stripe, and Google APIs directly in the generated code. The BOLA vulnerability combined credential exposure with context exposure in a single access event.

Incident 7 — deployment and cloud integration: Vercel and Context.ai.

The Vercel breach in April 2026 traced back to a Lumma Stealer infection on a Context.ai employee's machine in February 2026. Context.ai is a third-party AI productivity tool. A Vercel employee had connected it to their enterprise Google Workspace account with full OAuth permissions. The breach did not start at Vercel. It started two hops away.

Figure 10: *How a third-party application led to breaching an organization's customers.*

The attack chain:

1. In February 2026, a Lumma Stealer infection compromised a Context.ai employee's machine, harvesting stored OAuth tokens
2. A Vercel employee had connected Context.ai to their enterprise Google Workspace account with broad OAuth permissions, creating a trust relationship between the two organizations
3. Using the harvested OAuth token, the attacker accessed the Vercel employee's Google Workspace account
4. From Google Workspace, the attacker pivoted into Vercel's internal systems
5. Environment variables not marked as sensitive across a subset of customer projects were reached and exfiltrated, including API keys, npm tokens, GitHub tokens, and source code

No direct relationship between the affected Vercel customers and Context.ai was required. The OAuth trust relationship created the pathway.

This is what operating in a SaaS-dependent development environment means in practice. A well-run security program can do everything right internally and still inherit the blast radius of a vendor it depends on. Vercel did not fail at basic security hygiene. It was downstream of a compromise that originated two hops away. Planning for breaches in SaaS and PaaS dependencies is not a contingency for worst-case scenarios. It is a baseline operating assumption for any organization running a modern development stack. Driving in snow means accepting that your exposure is partly determined by how well everyone else on the road drives.

The blast radius reached Vercel customers with no direct relationship to Context.ai, through a single OAuth trust chain that connected a vendor's compromised employee machine to a customer's production environment variables.

3. The Connecting Pattern

Different groups. Different techniques. Different layers. One consistent insight.

The development environment contains assets distributed across every layer: credentials, signing keys, API tokens, environment variables. But it is also, collectively, a map. The CI/CD configuration documents the pipeline. The AI chat history documents the architecture. The OAuth grants

document the trust relationships. The dependency graph documents what the system trusts. That map is not locked in a vault. It is distributed across six layers, each with different ownership, different monitoring, and different security controls.

And through the trust relationships embedded in those layers, the development environment is a set of live pathways into production systems. Environment variables reach databases. OAuth grants reach cloud accounts. Pipeline tokens sign releases. The development environment does not just describe production. In many cases, it connects to it directly.

The attacks between October 2025 and April 2026 did not target a single high-value asset. They targeted the map and the pathways simultaneously, across every layer where they existed. That is the pattern.

4. Why the Standard Playbook Doesn't Fully Apply

The incident response playbook was designed for bounded compromise. Detect the intrusion, contain it, rotate the affected credentials, remediate, close the ticket. That sequence works when compromise is an asset problem: a credential was stolen, you rotate it, the stolen version is now useless.

The development environment is not bounded in that way.

Rotating credentials addresses the asset dimension of a compromise. It does not address the map dimension: every system those credentials authenticated to, every downstream trust relationship that extended from that access point, every pathway the attacker now understands. If the attacker read the architecture or accessed connected systems before the rotation, they still hold that map. The keys changed. The map did not.

The trust graph extends beyond what credential rotation covers, and the Trivy incident is the clearest proof available. Aqua Security detected the initial intrusion, rotated credentials, and followed the playbook correctly. The rotation was incomplete because the problem was not a list of credentials to rotate. It was a graph: every credential that had been accessible, every system that credential had authenticated to, every downstream trust relationship that extended from that access point. Rotating 95 percent of the credentials does not mean 95 percent containment. It means the attacker retains whatever access the remaining 5 percent provides, with full context about what it connects to.

Remediation in a connected environment is a graph traversal problem, not a checklist.

A separate tension has no clean resolution: supply chain security guidance says pin dependencies to immutable SHAs, to avoid version tags that can be silently force-pushed. Vulnerability management guidance says patch fast. Those two imperatives are in structural conflict. A pinned dependency resists silent tag manipulation but requires deliberate action to update, which slows the response to legitimate vulnerability disclosures. No current framework cleanly resolves this.

5. What Closing the Blast Radius Actually Requires

The question after a development environment compromise is not only "what did they take." It is "what do they now know, and what can they reach from here."

Those are different questions and they require different work to answer.

The first question is incident response: inventory what was accessed, rotate what was exposed, notify who needs to be notified. Most organizations have a process for this.

The second question requires tracing the full layer topology from the point of compromise outward: which credentials were within reach, which chat histories and pipeline configurations were accessible, what those credentials authenticated to across IAM roles and cloud providers, what the OAuth grants connected, what the pipeline touched while it was running malicious code. That work determines whether containment is real or whether the attacker still holds a map with working pathways into production.

Two controls limit how far the blast radius extends.

The first is credential scoping. The credential surface in a modern development environment spans human and non-human identities: developer tokens, service accounts, pipeline credentials, OAuth grants, API keys. Scoping those credentials to least privilege limits how far an attacker can move through the graph when any one of them is compromised.

Developer and DevOps identities belong explicitly in this picture. Their tokens sign releases. Their machines authenticate to cloud providers. Their pipeline access reaches production. The access they carry is production-grade. Hardening the developer environment without degrading the productivity that makes it valuable is a genuine design challenge. It is not, however, an optional one.

The second is patch velocity calibrated to blast radius, not just to CVSS score. The attack chain a given vulnerability enables, and what that chain reaches, determines how fast the fix needs to move. A critical CVE in an isolated library with no privileged access is a different problem from a moderate CVE in something that sits inside CI/CD pipelines with access to cloud credentials and signing keys. Blast radius is the variable that determines urgency.

The organizations that contained these incidents fastest were the ones that already knew what their development environment connected to before the attacker did. That knowledge is not a product of incident response. It is a prerequisite for it.

6. The Development Environment Was Always This Complex. The Stakes Keep Getting Higher.

The six layers described in this piece are not new. The AI toolchain layer is genuinely new, but everything else is familiar terrain to any practitioner who has run a CI/CD pipeline or managed a software supply chain program. SLSA and CNCF have been mapping this topology for years.

What changed between October 2025 and April 2026 is not the terrain. It is the systematic, multi-layer targeting of that terrain by attackers who clearly understood its structure. GlassWorm at the IDE layer. ForceMemo at the SCM layer. Trivy at CI/CD. Axios and LiteLLM at the dependency ecosystem. Lovable at the AI toolchain. Vercel at the cloud integration layer. Seven attacks across six layers in six months.

The development environment was always a map with live pathways into production. It was always a collection of assets with connections between them that compounded their individual value. What these attacks revealed is that adversaries have started treating it that way.

The question for security programs is not whether this will happen again. It is whether their threat model will be ready when it does.

References

1. Koi Security. "GlassWorm: First Self-Propagating Worm Using Invisible Code Hits OpenVSX Marketplace." October 20, 2025. https://www.koi.ai/incident/live-updates-glassworm-first-self-propagating-worm-using-invisible-code-hits-openvsx-and-vscode-marketplaces
2. Socket Research Team. "GlassWorm Supply-Chain Attack Abuses 72 Open VSX Extensions." March 13, 2026. https://socket.dev/blog/open-vsx-transitive-glassworm-campaign
3. The Hacker News. "GlassWorm Attack Uses Stolen GitHub Tokens to Force-Push Malware Into Python Repos." March 21, 2026. https://thehackernews.com/2026/03/glassworm-attack-uses-stolen-github.html
4. SecurityWeek. "ForceMemo: Python Repositories Compromised in GlassWorm Aftermath." March 16, 2026. https://www.securityweek.com/forcememo-python-repositories-compromised-in-glassworm-aftermath/
5. KENSAI, “ForceMemo Hijacks 400+ Python Repos.” March 17, 2026. https://kensai.app/blog/2026-03-18-glassworm-forcememo-eu-sanctions-apple-webkit-leaknet-deno-font-ai-evasion
6. Google Threat Intelligence Group. "North Korea-Nexus Threat Actor Compromises Widely Used Axios NPM Package." April 1, 2026. https://cloud.google.com/blog/topics/threat-intelligence/north-korea-threat-actor-targets-axios-npm-package
7. SOCRadar. "Axios npm Hijack 2026: IOCs, Impact and Remediation." April 2026. https://socradar.io/blog/axios-npm-supply-chain-attack-2026-ciso-guide/
8. Huntress. "Supply Chain Compromise of axios npm Package." March 31, 2026. https://www.huntress.com/blog/supply-chain-compromise-axios-npm-package
9. Aqua Security. "Trivy Supply Chain Attack: What You Need to Know." March 20, 2026. https://www.aquasec.com/blog/trivy-supply-chain-attack-what-you-need-to-know/
10. GitHub Security Advisory GHSA-69fq-xp46-6x23. https://github.com/aquasecurity/trivy/security/advisories/GHSA-69fq-xp46-6x23
11. CISA. "CVE-2026-33634 Added to Known Exploited Vulnerabilities Catalog." March 26, 2026. https://www.cisa.gov/news-events/alerts/2026/03/26/cisa-adds-one-known-exploited-vulnerability-catalog
12. Microsoft Security Blog. "Detecting, Investigating, and Defending Against the Trivy Supply Chain Compromise." March 24, 2026. https://www.microsoft.com/en-us/security/blog/2026/03/24/detecting-investigating-defending-against-trivy-supply-chain-compromise/
13. Help Net Security. "CISA Sounds Alarm on Trivy Supply Chain Compromise." March 27, 2026. https://www.helpnetsecurity.com/2026/03/27/cve-2026-33017-cve-2026-33634-exploited
14. Arctic Wolf. "TeamPCP Supply Chain Attack Campaign." March 2026. https://arcticwolf.com/resources/blog/teampcp-supply-chain-attack-campaign-targets-trivy-checkmarx-kics-and-litellm-potential-downstream-impact-to-additional-projects/
15. CyberKendra. “Lovable Left Thousands of Projects Exposed for 48 Days.” April 22, 2026. https://www.cyberkendra.com/2026/04/lovable-left-thousands-of-projects.html
16. Vercel. Official incident disclosure. April 2026. https://vercel.com/kb/bulletin/vercel-april-2026-security-incident
17. GitGuardian. "State of Secrets Sprawl 2025." https://www.gitguardian.com/state-of-secrets-sprawl-report-2025
18. Synopsys. "Open Source Security and Risk Analysis Report 2025." https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html
19. SLSA Framework. https://slsa.dev
20. Cloud Native Computing Foundation. https://cncf.io
21. Averlon. "CVE-2026-33634: Responding to the Trivy and LiteLLM Supply Chain Attacks." March 2026. https://www.averlon.ai/blog/cve-2026-33634-trivy-and-litellm-supply-chain-attacks
22. Averlon. "What the Vercel / Context.ai Incident Means for You." April 2026. https://www.averlon.ai/blog/what-the-vercel-context-ai-incident-means-for-you