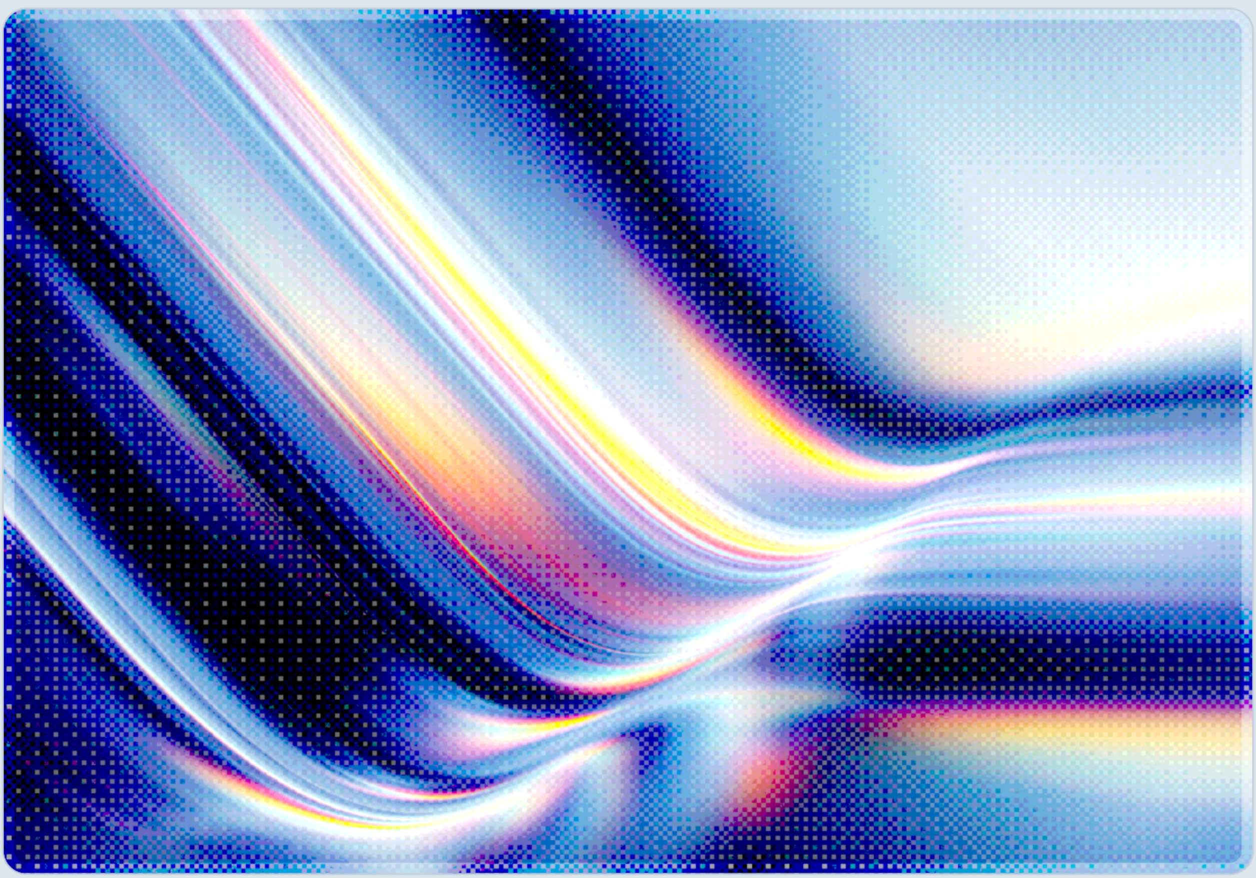
CSAI Foundation | Cloud Security Alliance

GitHub CVE-2026-3854: Push-Option Injection RCE on GHES

AI-Augmented Reverse Engineering Surfaces Critical Push Pipeline Flaw; 88% of GHES Instances Unpatched at Disclosure

2026-04-28

AI-assisted Rapid Research



CSAI cloud security alliance®

This document was generated with AI assistance and has not undergone official CSA review and approval processes.

Key Takeaways

- CVE-2026-3854 is a critical command injection vulnerability (CWE-77) in GitHub's `babeld` git proxy that allowed any authenticated user with push access to a single repository to execute arbitrary commands on GitHub's backend infrastructure using nothing more than a standard `git push` invocation. The flaw received CVSS v3.1 8.8 (HIGH) and CVSS v4.0 8.7 (HIGH) base scores [1][2][3].
- The vulnerability affected GitHub.com, GitHub Enterprise Cloud (including Data Residency and Enterprise Managed Users variants), and GitHub Enterprise Server (GHES) versions 3.14 through 3.19. GitHub deployed a fix to GitHub.com roughly seventy-five minutes after receiving the report on March 4, 2026 — a response GitHub itself characterizes as "less than two hours" — and released GHES patches on March 10, 2026. At public disclosure on April 28, 2026, Wiz reported that approximately 88% of internet-visible GHES instances still ran unpatched releases [3][4][5].
- On GitHub.com, successful exploitation would have provided cross-tenant code execution on shared storage nodes, granting filesystem access to repositories belonging to other users and organizations on the same node — a blast radius spanning millions of repositories regardless of owner [3][4][5].
- The root cause was a sanitization gap in how `babeld` copied user-supplied push option values verbatim into a semicolon-delimited internal `X-Stat` header. Researchers chained three injected fields — `rails_env`, `custom_hooks_dir`, and `repo_pre_receive_hooks` — to escape sandboxing, redirect the hook resolver, and execute attacker-controlled binaries as the `git` service user [3][6].
- The Wiz Research team discovered the bug using AI-augmented reverse engineering of GitHub's closed-source binaries via the IDA MCP integration, which Wiz characterizes as one of the first critical vulnerabilities in proprietary compiled software surfaced by an AI-assisted analysis workflow [3].
- GitHub's forensic review found no evidence of pre-disclosure exploitation by any party other than the reporting researchers; nevertheless, GHES administrators should patch immediately to one of the fixed branches (3.14.25, 3.15.20, 3.16.16, 3.17.13, 3.18.7, or 3.19.4) and audit `/var/log/github-audit.log` for push operations whose push option values contain semicolons as a retroactive indicator of exploitation attempts [2][4][5].

Background

GitHub is among the most widely used code-hosting platforms in the global software supply chain, and its self-hosted variant, GitHub Enterprise Server, is commonly deployed in environments where SaaS use is restricted, including regulated industries, classified environments, and air-gapped networks. A remote code execution vulnerability that traverses the platform's git push pipeline therefore poses a multi-tenant risk on the SaaS side and a sovereign-infrastructure risk on the self-hosted side. CVE-2026-3854, disclosed publicly on April 28, 2026, was a vulnerability of exactly this character: a single crafted `git push` could have compromised the receiving instance from any account with write access to any repository on it [1][3].

The vulnerability was reported to GitHub on March 4, 2026, by the Wiz Research team through GitHub's bug bounty program. According to GitHub's incident timeline, the company validated the report within forty minutes of receipt and deployed a fix to GitHub.com by 7:00 p.m. UTC the same day, roughly seventy-five minutes after notification — a response GitHub itself characterizes as "less than two hours" [4]. CVE-2026-3854 was assigned and GHES patches were published on March 10, 2026 [1][2]. GitHub stated the finding "will receive one of the highest rewards in the history of our Bug Bounty program," reflecting the unusual combination of pre-authorization simplicity and unrestricted post-exploitation reach across the GitHub.com fleet [4].

A distinguishing feature of this disclosure is its discovery methodology. The Wiz team identified the bug using AI-augmented reverse engineering of GitHub's closed-source binaries through the IDA MCP integration; the dedicated subsection later in this note discusses what this approach signals for the broader pace of platform-vulnerability discovery [3].

Security Analysis

Vulnerability Mechanics

Git's push protocol allows clients to attach key-value strings to a push using the `-o` (push option) flag. Push options carry hints about review policies, deployment targets, or CI behavior, and they are passed by the server-side git infrastructure to internal services that act on those hints. In GitHub's architecture, push option values traverse `babeld`, the SSH-fronting git proxy, before being forwarded as fields in a semicolon-delimited internal header named `X-Stat`. The `gitrpcd` RPC server downstream then parses `X-Stat` into authoritative push-time security metadata – including the runtime environment to execute under, the directory in which to look for hook binaries, and the list of pre-receive hooks to run [3][6].

The defect was that `babeld` copied the raw push option value into the `X-Stat` header without escaping or rejecting the semicolon character that delimited fields in that header. Any push option value containing a semicolon could therefore terminate its own field and inject one or more new fields. The downstream parser used last-write-wins semantics for duplicate keys, meaning that an attacker-injected duplicate of a legitimate field would silently override the value supplied by the trusted control plane [3].

The Wiz researchers chained three field injections to convert this primitive into reliable remote code execution. First, they injected a non-production value for `rails_env`, which moved request handling onto a code path that was never reached on github.com under normal operation and that bypassed the sandbox normally constraining hook execution. Second, they injected `custom_hooks_dir` to redirect the hook resolver to an attacker-controlled location. Third, they injected `repo_pre_receive_hooks` with a crafted entry whose path traversed out of the expected hook directory and pointed at an arbitrary executable. Once the push completed, the pre-receive hook stage executed the attacker's binary as the `git` service user, yielding native code execution on GitHub's backend [3][6]. The entire exploit was triggered with a single command of the form `git push -o "<crafted payload>" origin <branch>`.

Blast Radius

The blast radius differed materially between GitHub.com and self-hosted GitHub Enterprise Server, and both deserve independent treatment.

On GitHub.com, the `git` service user on a storage node has filesystem access to every repository hosted on that node. GitHub's multi-tenant architecture co-locates customer repositories on shared storage infrastructure, and the on-disk layout makes neighboring tenants directly readable by any process running with `git` privileges on the same node. Although GitHub maintains organizational and access-control boundaries at the application layer, code execution at the storage tier would have rendered those application-layer boundaries ineffective for preventing cross-tenant read access. GitHub and Wiz both characterize the resulting blast radius as cross-tenant exposure to "millions of repositories" on each compromised node, regardless of the organization or user that owned them [3][4][5]. In practice this means a single bug-bounty proof of concept, had it been weaponized rather than reported, could have read source code, secrets, and CI configuration belonging to organizations entirely unrelated to the attacker's account.

On GitHub Enterprise Server, the blast radius is bounded by the customer's deployment but still represents complete compromise of that deployment, including all hosted repositories, stored credentials and webhook secrets, administrative configuration, and any infrastructure the appliance can reach. For organizations that use GHES as the centerpiece of their software supply chain — including those that deploy GHES specifically to satisfy data-residency or air-gap requirements — successful exploitation constitutes a top-level compromise of source code integrity for the enterprise [1][2]. Operators that run multi-tenant GHES deployments on behalf of multiple internal business units should treat cross-business-unit exposure as in scope: a successful exploit grants `git`-privileged access across all repositories on the appliance, regardless of the business-unit ownership boundaries enforced at the application layer.

AI-Augmented Discovery as Methodological Signal

The methodology used to find CVE-2026-3854 is itself a meaningful data point for security architects evaluating vulnerability emergence rates over the next several quarters. The Wiz Research team has publicly attributed the discovery to AI-assisted reverse engineering using the IDA MCP integration applied to GitHub's proprietary binaries, and characterizes the finding as one of the first critical vulnerabilities in closed-source compiled software discovered through an AI-assisted workflow [3]. The relevance for defenders is twofold. First, vulnerabilities in proprietary backend services that interconnect through internal protocols — exactly the class to which CVE-2026-3854 belongs — have historically been protected, in significant part, by the high cost of manual reverse engineering. Recent AI-augmented analysis efforts, including the one that produced this disclosure, suggest that cost is declining. Second, the same AI-assisted techniques are accessible to less constructive parties, which compresses the expected interval between the introduction of a defect into a closed-source codebase and its eventual discovery, regardless of whether the discoverer is a defender, a bug-bounty researcher, or an adversary.

This dynamic does not change the controls organizations should apply to GitHub or to similar platforms, but it does change the cadence at which they should expect those platforms to disclose critical findings. Recent CSA research notes documenting AI-driven vulnerability discovery point to the same trend now showing up in the proprietary code that runs the source code platform itself.

Exploitation Status and Detection

GitHub stated that the unusual structure of the exploit aided its forensic investigation: the malicious code path it forced the server to take was not exercised by any normal git operation on GitHub.com, so every occurrence of that code path in the relevant logs mapped to the Wiz researchers' own testing activity, no other accounts triggered it, and no customer data was accessed, modified, or exfiltrated in the pre-disclosure window [4]. As of public disclosure, no in-the-wild exploitation has been reported, no public proof-of-concept has been released, and the vulnerability is not listed in CISA's Known Exploited Vulnerabilities catalog. The EPSS score at disclosure was 0.4 at the 60.2nd percentile, placing it in the upper 40% of vulnerabilities by short-term exploitation likelihood [8].

For GHES administrators, GitHub recommends inspecting `/var/log/github-audit.log` for push operations whose push option values contain semicolon characters. Because legitimate push option usage rarely if ever contains semicolons, this signature is high-precision for retrospective detection of exploitation attempts on appliances that ran a vulnerable release prior to upgrade [4]. Organizations that capture appliance audit logs into their SIEM should run this search across the full retention window covering March 10, 2026 to the date the appliance was patched.

Patch Posture and the GHES Patch Lag

The version table below summarizes the affected and fixed releases. Any GHES instance below the corresponding fixed version on its branch is vulnerable.

GHES Branch	Vulnerable Versions	Fixed Version
3.14.x	3.14.0 – 3.14.24	3.14.25
3.15.x	3.15.0 – 3.15.19	3.15.20
3.16.x	3.16.0 – 3.16.15	3.16.16
3.17.x	3.17.0 – 3.17.12	3.17.13
3.18.x	3.18.0 – 3.18.6	3.18.7
3.19.x	3.19.0 – 3.19.3	3.19.4

Wiz reported that approximately 88% of internet-visible GHES instances were still on a vulnerable release at public disclosure on April 28, 2026, despite patches having been available since March 10 [3] [5]. This patch lag appears consistent with patterns observed in prior GHES vulnerability disclosures and likely reflects the operational reality of GHES as appliance software: customers control their own upgrade cadence, and GHES is frequently deployed in environments with strict change-control processes that resist out-of-cycle patching. The implication for incident responders is that the population of exploitable targets is likely to remain large for weeks to months following disclosure, and the disclosure itself materially lowers the cost of weaponizing the bug for any party that begins from the published technical write-ups.

Recommendations

Immediate Actions

GHES administrators should treat CVE-2026-3854 as a critical, time-sensitive patch. The remediation is unambiguous: upgrade the appliance to the relevant fixed version (3.14.25, 3.15.20, 3.16.16, 3.17.13, 3.18.7, or 3.19.4) using GitHub's standard appliance upgrade procedure [2][4]. There is no documented configuration-only mitigation that fully neutralizes the vulnerability while keeping the push pipeline operational. Restricting push access to a smaller set of trusted users is a partial compensating control, but because any user with push access to any repository on the appliance — including a repository they create themselves — can trigger the exploit, the practical population of authorized triggers is rarely small enough to provide meaningful defense.

In parallel with patching, administrators should audit `/var/log/github-audit.log` for push operations whose push option values contain semicolons across the full retention window from March 10, 2026 through the date of the appliance upgrade. Any matches warrant treatment as suspected exploitation attempts: preserve the appliance state for forensic review, rotate credentials reachable from the appliance (including webhook secrets, deploy keys, and any cached cloud credentials), and review process execution and outbound network logs from the appliance host for indicators of post-exploitation activity. Customers with managed GHES deployments should confirm with their managed-service provider that the upgrade has been applied and that audit log review has been performed.

GitHub.com, GitHub Enterprise Cloud, GitHub Enterprise Cloud with Data Residency, and GitHub Enterprise Cloud with Enterprise Managed Users are all patched at the platform level and require no customer action [4]. Customers using these services who maintain access logging through GitHub's audit log streaming capabilities may wish to perform the same semicolon-search review for completeness, but no upgrade is required.

Short-Term Mitigations

Organizations should use this disclosure as the trigger for a structured review of how GHES appliances are positioned within their change management and vulnerability management programs. GHES should be classified as Tier 0 infrastructure for the purposes of patching SLAs, given that compromise of a GHES instance constitutes compromise of the source code integrity foundation of the enterprise. Patch

SLAs of seven days or less for critical-severity GHES advisories are appropriate; the observed 49-day gap during which a majority of internet-visible instances remained unpatched falls short of that bar [3] [5].

Network exposure of GHES appliances should be reviewed against the principle of least exposure. GHES instances accessible from the public internet face a population of opportunistic attackers that grows substantially during the disclosure window of any critical vulnerability. Where the operational requirement permits, GHES should be reachable only from the corporate network, from a defined allowlist of partner networks, or through a zero-trust access broker enforcing device posture and identity, rather than directly from the internet. This positioning does not eliminate the risk of insider-triggered exploitation, but it materially reduces the population of accounts capable of attempting exploitation and removes the appliance from opportunistic scanning.

For organizations that operate GHES appliances on behalf of others — managed service providers, internal platform engineering teams running shared developer infrastructure, and similar arrangements — the disclosure should prompt a review of the customer-communication process for GHES advisories. The 49-day patch lag observed in this incident reflects, in part, slow propagation of advisory information through downstream operator channels. Documented advisory pipelines that route GitHub advisories directly to the on-call team responsible for upgrade execution, with explicit SLA commitments to customer organizations, are a structural fix.

Strategic Considerations

CVE-2026-3854 is best understood not as an isolated GitHub defect but as a worked example of a broader pattern: code execution vulnerabilities in the source code platform are software supply chain incidents at the largest possible scope, and the discovery economics of those vulnerabilities are shifting in favor of any party that can deploy AI-augmented reverse engineering at scale. Organizations whose security architecture has implicitly assumed that proprietary platform binaries are inaccessible to attackers should revisit that assumption.

The cross-tenant reach observed in the GitHub.com case is also worth holding up for architectural reflection. SaaS providers that host customer data on shared infrastructure — which is to say, most large SaaS providers — generally maintain tenant boundaries at the application layer and rely on operational controls to prevent code execution at the storage or platform tier. Where those controls fail, even a brief code execution window can yield read access to other tenants. CSA's Cloud Controls Matrix and AI Controls Matrix both call for tenant isolation controls at multiple layers; CVE-2026-3854 is a reminder that these controls must include defense-in-depth at the storage and process tier, not only at the request and authentication tier.

Finally, the AI-augmented discovery methodology used in this case has implications for how organizations should think about the velocity of platform vulnerability disclosure over the next several quarters. As AI-assisted reverse engineering tools become broadly available — including through open MCP integrations such as the one used by the Wiz team — the cost of identifying defects in compiled, closed-source software falls. Defenders should plan for the possibility of more frequent critical disclosures from major platform vendors, potentially compressed exploit-development timelines following those disclosures, and reduced reliance on security-through-obscurity assumptions in proprietary backends.

CSA Resource Alignment

CVE-2026-3854 maps onto multiple CSA frameworks that organizations should consult when contextualizing the response.

The CSA AI Controls Matrix (AICM) [9], which extends and supersedes the Cloud Controls Matrix (CCM) [10] for AI-relevant risks, addresses this vulnerability across several control domains. The Vulnerability and Patch Management domain governs the SLA-bound remediation of disclosed defects; the Threat and Vulnerability Management domain governs the detection of post-exploitation activity through audit log review; and the Application and Interface Security domain governs the input validation expectations that the `babeld` push option handler failed to meet. Organizations using AICM as their primary control framework should map their GHES appliance to each of these control families and confirm that documented control implementations cover the appliance specifically, not only general-purpose application infrastructure.

CSA's Software Supply Chain Security guidance [12] is directly applicable. Compromise of a source code platform — whether SaaS or self-hosted — represents a high-severity software supply chain incident, since it grants the attacker write access to the artifacts that downstream build, test, and deployment systems trust. CSA's published guidance on signed commits, protected branches, branch protection rules, and reproducible builds becomes more important, not less, as platform-level RCE vulnerabilities continue to surface. Organizations that have not yet adopted commit signing and branch protection should treat this disclosure as additional motivation for that work.

CSA's MAESTRO threat modeling framework for agentic AI [11] is relevant in two respects. First, the AI-augmented reverse engineering methodology that produced this discovery is itself an instance of agentic AI applied to vulnerability research, and MAESTRO's data-and-model-layer threat categories help organizations think through both the offensive and defensive uses of such pipelines. Second, MAESTRO's emphasis on the integration and tool layer — where AI agents interact with external systems through protocols such as MCP — applies directly to the `babeld` defect's root cause: insufficient sanitization at a trust boundary where externally controlled key-value data is consumed by an internal protocol.

Finally, CSA's Zero Trust guidance [13] recommends that high-value internal services authenticate, authorize, and integrity-check every inter-service request rather than implicitly trusting cross-component metadata. The `X-Stat` header pattern at the heart of CVE-2026-3854 — a structured metadata channel constructed from partially user-controlled input and consumed by a downstream service that treats its fields as authoritative — is a clear example of the kind of internal-trust assumption that CSA's Zero Trust guidance explicitly cautions against. Organizations operating their own platforms with similar internal protocol patterns should treat this disclosure as a prompt to audit any analogous metadata-passing channels in their own architecture.

References

[1] National Vulnerability Database. "CVE-2026-3854 Detail." NIST NVD, March 10, 2026.

[2] OpenCVE. "CVE-2026-3854 – Vulnerability Details." OpenCVE, March 2026.

[3] Wiz Research. "GitHub RCE Vulnerability: CVE-2026-3854 Breakdown." Wiz Blog, April 28, 2026.

[4] Alexis Wales / GitHub Security. "Securing the git push pipeline: Responding to a critical remote code execution vulnerability." The GitHub Blog, April 28, 2026.

[5] The Hacker News. "Researchers Discover Critical GitHub CVE-2026-3854 RCE Flaw Exploitable via Single Git Push." The Hacker News, April 28, 2026.

[6] Cybersecurity News. "Critical GitHub.com and Enterprise Server RCE Vulnerability Enables Full Serve r Compromise." Cybersecurity News, April 28, 2026.

[7] OffSeq Threat Radar. "CVE-2026-3854 – CWE-77 Improper Neutralization in GitHub Enterprise Ser ver." OffSeq, April 2026.

[8] Wiz Vulnerability Database. "CVE-2026-3854 Impact, Exploitability, and Mitigation Steps." Wiz, April 2026.

[9] Cloud Security Alliance. "AI Controls Matrix (AICM)." Cloud Security Alliance, 2025.

[10] Cloud Security Alliance. "Cloud Controls Matrix (CCM)." Cloud Security Alliance, 2024.

[11] Cloud Security Alliance. "MAESTRO: Agentic AI Threat Modeling Framework." Cloud Security Alliance, February 6, 2025.

[12] Cloud Security Alliance. "Software Transparency: Securing the Digital Supply Chain." Cloud Security Alliance, accessed April 2026.

[13] Cloud Security Alliance. "Zero Trust Guidance." Cloud Security Alliance, accessed April 2026.