INTELLIGENCE BRIEFING
Security Command Center
TLP:CLEAR
2026-05-01 18:54 UTC

BufferZoneCorp Supply Chain Campaign Plants Sleeper Packages Across Ruby and Go Ecosystems to Compromise CI Pipelines

THREAT CAMPAIGN | **HIGH** | CVSS 7.5

SCC Item ID	SCC-CAM-2026-0254
Type	Threat Campaign
Severity	HIGH
CVSS Base Score	7.5
Affected Products	RubyGems ecosystem, Go modules ecosystem, GitHub Actions workflows, AWS credential stores, npmrc files, GitHub CLI environments, SSH key stores, CI/CD runners
Published	2026-05-01T05:43:00
Discovery Source	Rss

Executive Summary

A threat actor using the GitHub identity 'BufferZoneCorp' published malicious packages across the Ruby and Go open-source ecosystems, embedding credential-harvesting payloads that execute silently during routine dependency installation in CI/CD pipelines. Any development or build environment that installed affected packages before their removal should be treated as presumed compromised pending forensic validation, with AWS credentials, SSH keys, GitHub Actions secrets, and API tokens at risk of exfiltration. The business risk is significant: stolen CI credentials can enable attackers to pivot into cloud infrastructure, poison build artifacts, and establish persistent access across production environments.

Technical Analysis

Threat actor 'BufferZoneCorp' staged malicious RubyGems and Go modules designed to execute during package installation (install hooks / init functions), requiring no additional user interaction. The campaign used a dual-track strategy: some packages carried immediately active credential-harvesting payloads; others were delayed-activation packages, benign at install time but designed for later activation via a trigger mechanism [specific activation conditions not confirmed in available sources]. Targeted assets include environment variables, SSH keys (~/.ssh/), AWS credential files (~/.aws/credentials), .npmrc tokens, GitHub CLI tokens, and GitHub Actions secrets accessible within runner contexts. Exfiltration used standard HTTP/S (T1071.001). Persistence mechanisms were embedded for post-install execution (T1547, T1053). The packages impersonated legitimate developer tooling (T1608.001). CWEs: CWE-506 (embedded malicious code),

CWE-494 (download of code without integrity check), CWE-312 (cleartext storage of sensitive information), CWE-829 (inclusion of functionality from untrusted control sphere). MITRE coverage: T1195.001 (supply chain compromise), T1552.001 (credentials in files), T1552.004 (private keys), T1528 (steal application access tokens), T1554 (compromise client software binary), T1098.004 (SSH authorized keys). Packages have been removed from RubyGems.org and blocked by the Go module proxy (sum.golang.org). No CVE has been assigned. Qualitative severity is rated High based on operational scope and credential exposure risk; CVSS metrics are not available from vendor sources. Environments that installed affected packages before removal should be treated as presumed compromised and require credential rotation and forensic audit.

Action Checklist

1. Step 1: Containment, Identify all CI/CD runners, developer workstations, and build environments that executed a gem install or go get/go mod tidy within the window the packages were live. Isolate those environments from production networks and cloud credential stores immediately. Capture credential files and access logs for forensic analysis before rotation. Revoke all AWS IAM credentials, GitHub personal access tokens, SSH keys, and .npmrc tokens accessible from those environments within 4 hours of isolation.
2. Step 2: Detection, Search CI/CD logs and package manager audit logs for any installation of packages published under the 'BufferZoneCorp' namespace. [Specific package names: confirm and list if available.] Review GitHub Actions workflow logs for unexpected outbound HTTP/S connections from runner steps. Check ~/.aws/credentials, ~/.ssh/known_hosts write events, and .npmrc modification timestamps for anomalies during the exposure window. Query SIEM for outbound connections from build runners to unknown external IPs or domains around installation time (T1071.001 behavioral pattern).
3. Step 3: Eradication, Remove all packages from 'BufferZoneCorp' from Gemfiles, go.mod, and lock files. Rebuild all affected containers and runner images from a known-clean base. Rotate every credential class confirmed accessible in compromised environments: AWS IAM keys, GitHub tokens, SSH keypairs, .npmrc auth tokens, and any secrets injected via CI environment variables. Re-scan rebuilt environments with a dependency integrity tool (e.g., bundler-audit for Ruby, govulncheck for Go) before returning to service.
4. Step 4: Recovery, After credential rotation, validate that new credentials are functioning and that no unauthorized IAM policies, SSH authorized_keys entries, or OAuth app authorizations were added during the exposure window. Review AWS CloudTrail, GitHub audit logs, and SSH access logs for lateral movement or persistence artifacts introduced using stolen credentials. Monitor build pipelines for anomalous artifact checksums or unexpected dependency additions for a minimum of 30 days post-remediation.
5. Step 5: Post-Incident, Evaluate current controls against CWE-494 and CWE-829: implement or enforce dependency pinning with hash verification (e.g., Gemfile.lock with checksum enforcement, go.sum verification). Adopt a software composition analysis (SCA) tool integrated into the CI pipeline to flag new or unrecognized package publishers. Review the principle of least privilege for CI runner credential scopes; runners should not have access to production AWS credentials or broad GitHub token scopes. Map control gaps to NIST SP 800-161 (supply chain risk management) and NIST CSF Respond/Recover functions.

IR / Forensic Enrichment

Triage Priority	IMMEDIATE
Escalation Criteria	Escalate to CISO, legal, and breach notification counsel immediately if AWS CloudTrail or GitHub audit logs confirm that stolen credentials were used to access, exfiltrate, or modify production data stores, customer PII, PHI, or regulated data — or if evidence of lateral movement from CI runners into production VPCs or cloud accounts is detected, as this may trigger breach notification obligations under GDPR, CCPA, or HIPAA depending on data classification.
Recovery Notes	After credential rotation, do not restore CI pipelines to production use until AWS IAM Access Advisor confirms no API calls were made using the new credentials from unexpected source IPs, and until GitHub audit logs show no unauthorized collaborator or webhook additions in the post-rotation window. Monitor all build artifact registries (ECR, RubyGems private feeds, Go module proxies) for unexpected image pushes or package publications for a minimum of 30 days, as the BufferZoneCorp payload had access to credentials that could have been used to poison downstream artifacts or establish persistent publish access. Treat any artifact built or published during the exposure window as potentially tainted and re-build from source with verified dependency manifests before redistribution.
Forensic Artifacts	RubyGems bundler cache at ~/.gem/specs/ and /usr/local/bundle/cache/ — the installed .gem files for BufferZoneCorp packages will contain the actual credential-harvesting payload code; preserve as primary malware evidence before eradication. | Go module cache at $GOPATH/pkg/mod/cache/download/ — the downloaded module zip files and their corresponding go.sum hash entries for BufferZoneCorp modules; the zip contents contain the malicious Go source that executed during `go mod tidy` or `go get`. | GitHub Actions workflow run logs for all jobs in the exposure window — downloadable per-run via the GitHub API; these contain the stdout/stderr of gem install and go mod commands including any error output from the payload, and runner environment variable dumps if debug logging was enabled (ACTIONS_STEP_DEBUG=true). | AWS CloudTrail management event logs filtered for the IAM write event types CreateAccessKey, GetSecretAccessKey, AssumeRole, AttachUserPolicy, and PutRolePolicy originating from runner IP addresses or EC2 instance profiles during the exposure window — establishes whether harvested AWS credentials were used for reconnaissance or persistence. | File system write audit records for ~/.aws/credentials, ~/.ssh/known_hosts, ~/.ssh/authorized_keys, and .npmrc files from runner hosts — sourced from auditd logs (/var/log/audit/audit.log) or Linux inotify-based monitoring — with exact timestamps confirming when the BufferZoneCorp payload wrote to or read from these credential stores during package installation.

Per-Action IR Details

Step 1: Containment — Identify all CI/CD runners, developer workstations, and build environments that executed a gem install or go get/go mod tidy within the window the packages were live. Isolate those environments from production networks and cloud credential stores immediately. Revoke all AWS IAM credentials, GitHub personal access tokens, SSH keys, and .npmrc tokens accessible from those environments.

NIST Phase: Containment

Reference: NIST 800-61r3 §3.3 — Containment Strategy: isolate affected systems and preserve evidence before eradication to prevent further credential exfiltration from CI runners that ingested BufferZoneCorp packages.

Controls: NIST IR-4 (Incident Handling), NIST AC-2 (Account Management) — disable or revoke all IAM, GitHub PAT, and SSH credentials accessible from compromised runners, NIST SC-7 (Boundary Protection) — network-isolate affected CI runners from production AWS accounts and VPCs, CIS 5.3 (Disable Dormant Accounts) — treat all credentials exposed during the window as compromised and disable immediately, CIS 6.2 (Establish an Access

Revoking Process) — invoke the documented process to revoke GitHub PATs, AWS IAM keys, SSH keypairs, and .npmrc tokens sourced from affected environments

Compensating: For teams without automated CSPM or PAM tooling: run `aws iam list-access-keys --user-name ` and `aws iam delete-access-key` for each key accessible from affected runners; revoke GitHub PATs via `gh auth token` or the GitHub Settings > Developer Settings > PATs UI; remove SSH public keys from `~/.ssh/authorized_keys` on all hosts reachable from the runner; use `grep -r 'BufferZoneCorp' ~/.bundle/cache/ /tmp/ /home/runner/` to confirm package presence before network isolation. Firewall the runner host with `iptables -I OUTPUT -j DROP` or equivalent to prevent further exfiltration while investigation proceeds.

Evidence: BEFORE isolating, snapshot: (1) full contents of `~/.aws/credentials`, `~/.aws/config`, and any AWS_* environment variables exported in the runner session; (2) `~/.ssh/known_hosts` and `~/.ssh/authorized_keys` with `stat` timestamps to establish last-write time; (3) `.npmrc` files in home directory and project root with modification timestamps (`ls -la --full-time ~/.npmrc`); (4) running process list (`ps auxf`) and active network connections (`ss -tulnp` or `netstat -anp`) on runner hosts at time of isolation; (5) CI/CD runner job logs for the full execution window, including environment variable dumps if enabled — these may contain injected secrets that the payload harvested.

Step 2: Detection — Search CI/CD logs and package manager audit logs for any installation of packages published under the 'BufferZoneCorp' GitHub account. Review GitHub Actions workflow logs for unexpected outbound HTTP/S connections from runner steps. Check ~/.aws/credentials, ~/.ssh/known_hosts write events, and .npmrc modification timestamps for anomalies during the exposure window. Query SIEM for outbound connections from build runners to unknown external IPs or domains around installation time (T1071.001 behavioral pattern).

NIST Phase: Detection Analysis

Reference: NIST 800-61r3 §3.2 — Detection and Analysis: correlate package installation timestamps from RubyGems and Go module audit trails with outbound network activity and credential file write events to establish the scope of BufferZoneCorp payload execution across the CI environment.

Controls: NIST SI-4 (System Monitoring) — monitor CI runner process execution and network egress for payload behavioral indicators, NIST AU-6 (Audit Record Review, Analysis, and Reporting) — review GitHub Actions logs, RubyGems install logs, and Go module download logs for BufferZoneCorp package references, NIST AU-12 (Audit Record Generation) — verify that runner environments were generating audit records during the exposure window, CIS 8.2 (Collect Audit Logs) — ensure CI runner audit logs, GitHub Actions workflow logs, and AWS CloudTrail logs are available and cover the full exposure window, MITRE ATT&CK T1195.001 (Supply Chain Compromise: Compromise Software Dependencies and Development Tools) — pivot from package installation events to post-exploitation indicators, MITRE ATT&CK T1071.001 (Application Layer Protocol: Web Protocols) — identify C2 or exfiltration traffic over HTTP/S originating from runner processes spawned by gem or go toolchain

Compensating: Without a SIEM: (1) Parse RubyGems install logs at `~/.gem/specs/` and Bundler output logs — grep for 'BufferZoneCorp' across all CI job transcript files stored by your CI platform (GitHub Actions stores these as downloadable artifacts per workflow run). (2) For Go: inspect `$GOPATH/pkg/mod/cache/download/` directory for any module path containing 'bufferzonecorp' and cross-reference `go.sum` entries. (3) Use `auditd` or review `/var/log/audit/audit.log` for `SYSCALL` records showing `openat` or `write` on `~/.aws/credentials` or `~/.ssh/` during the installation window — filter with `ausearch -f /root/.aws/credentials --start `. (4) Extract outbound DNS queries from runner host using `tcpdump -i any -w capture.pcap port 53` replay and filter with `tshark -r capture.pcap -Y 'dns.qry.name'` to identify exfiltration domains. Apply the public Sigma rule for T1071.001 against any available syslog exports.

Evidence: BEFORE this step is complete, preserve: (1) GitHub Actions workflow run logs for every job in the exposure window — download via `gh run view --log` for each affected repository; (2) RubyGems bundler verbose output logs (`BUNDLE_VERBOSE=1`) or gem install stdout captures if stored by the CI platform; (3) Go module proxy logs from `GOPROXY` if a private or corporate proxy was in use, and `$GOPATH/pkg/mod/cache/download/sumdb/` entries for tampered packages; (4) DNS resolver logs or VPC Flow Logs showing connections from runner IP ranges to external destinations during `gem install` or `go mod tidy` execution steps; (5) File system modification timestamps on `~/.aws/credentials`, `.npmrc`, and `~/.ssh/known_hosts` — capture with `stat` or `find / -newer -ls` relative to a pre-installation baseline.

Step 3: Eradication — Remove all packages from 'BufferZoneCorp' from Gemfiles, go.mod, and lock files. Rebuild all affected containers and runner images from a known-clean base. Rotate every credential class confirmed accessible in compromised environments: AWS IAM keys, GitHub tokens, SSH keypairs, .npmrc auth tokens, and any secrets injected via CI environment variables. Re-scan rebuilt environments with a dependency integrity tool (e.g., bundler-audit for Ruby, govulncheck for Go) before returning to service.

NIST Phase: Eradication

Reference: NIST 800-61r3 §3.4 — Eradication: remove BufferZoneCorp malicious packages from all dependency manifests and lock files, destroy compromised runner images, and rotate the full credential surface accessible during the payload execution window before rebuilding CI infrastructure.

Controls: NIST SI-2 (Flaw Remediation) — remove malicious BufferZoneCorp packages from Gemfile.lock, go.sum, and go.mod and verify integrity of rebuilt images, NIST CM-3 (Configuration Change Control) — rebuild runner container images from a verified clean base image with documented change record, NIST IA-5 (Authenticator Management) — rotate all credential classes (AWS IAM, GitHub PAT, SSH keypairs, .npmrc tokens, CI environment secrets) confirmed in scope, NIST SI-7 (Software, Firmware, and Information Integrity) — use bundler-audit and govulncheck to verify dependency integrity of rebuilt environments before return to service, CIS 7.3 (Perform Automated Operating System Patch Management) — ensure rebuilt runner base images are fully patched before reintroduction, CIS 7.4 (Perform Automated Application Patch Management) — verify all Ruby gems and Go modules in rebuilt environments resolve from verified, non-BufferZoneCorp sources with checksum validation

Compensating: For teams without enterprise secret management: (1) Rotate AWS IAM keys via `aws iam create-access-key` then `aws iam delete-access-key --access-key-id ` and update all CI secret stores; (2) Generate new SSH keypairs with `ssh-keygen -t ed25519 -C 'ci-runner-replacement'` and remove old public keys from all `~/.ssh/authorized_keys` and GitHub Deploy Keys; (3) Run `bundle exec bundler-audit check --update` against the rebuilt Gemfile.lock to confirm no remaining vulnerable or malicious gems; (4) Run `govulncheck ./...` in the rebuilt Go module workspace; (5) Use `docker build --no-cache` to force a clean rebuild of all runner images, and verify the resulting image digest against your registry before use — store the expected digest in a text file in your repo as a manual integrity baseline.

Evidence: BEFORE eradicating, preserve forensic copies of: (1) The exact Gemfile.lock and go.sum files that were present when the malicious packages were installed — these establish the precise package versions and hashes of the BufferZoneCorp payload for later malware analysis; (2) A tarball of the gem cache directory (`~/.gem/` or `/usr/local/bundle/`) and Go module cache (`$GOPATH/pkg/mod/cache/`) from affected runners, preserving the actual malicious package binaries; (3) A memory dump or at minimum a full process tree capture (`ps auxf --forest > process_tree.txt`) from any runner that was live during payload execution, as the credential-harvesting payload may have been memory-resident; (4) All CI environment variable values (masked where possible) documented at time of compromise to establish the full credential scope accessible to the payload.

Step 4: Recovery — After credential rotation, validate that new credentials are functioning and that no unauthorized IAM policies, SSH authorized_keys entries, or OAuth app authorizations were added during the exposure window. Review AWS CloudTrail, GitHub audit logs, and SSH access logs for lateral movement or persistence artifacts introduced using stolen credentials. Monitor build pipelines for anomalous artifact checksums or unexpected dependency additions for a minimum of 30 days post-remediation.

NIST Phase: Recovery

Reference: NIST 800-61r3 §3.5 — Recovery: after credential rotation, verify environment integrity by auditing IAM policy changes, SSH authorized_keys modifications, and GitHub OAuth authorizations introduced during the BufferZoneCorp exposure window before restoring CI pipelines to production use.

Controls: NIST IR-4 (Incident Handling) — execute recovery phase actions consistent with documented IR plan, verifying containment and eradication effectiveness before restoration, NIST AU-6 (Audit Record Review, Analysis, and Reporting) — review AWS CloudTrail, GitHub audit logs, and SSH access logs for attacker persistence or lateral movement using stolen credentials, NIST AC-2 (Account Management) — audit for unauthorized IAM policies, OAuth app authorizations, and SSH authorized_keys entries created during the exposure window, NIST SI-4 (System Monitoring) — monitor rebuilt CI pipelines for anomalous artifact checksums, unexpected dependency additions, or unauthorized package publisher changes for 30 days, CIS 5.1 (Establish and Maintain an Inventory of Accounts) —

reconcile current IAM users, roles, GitHub collaborators, and SSH keys against pre-incident baseline to identify attacker-created persistence, MITRE ATT&CK T1098 (Account Manipulation) — hunt for new IAM policies, SSH keys, or GitHub collaborator additions that may have been introduced using credentials stolen by the BufferZoneCorp payload

Compensating: Without a SIEM: (1) Pull AWS CloudTrail events for the exposure window using `aws cloudtrail lookup-events --lookup-attributes AttributeKey=EventName,AttributeValue=CreateAccessKey --start-time --end-time ` and repeat for `AttachUserPolicy`, `CreateRole`, `AssumeRole`; (2) Query GitHub audit log via `gh api /orgs//audit-log?phrase=action:org.add_member+action:repo.add_collaborator` for the exposure window; (3) Diff current `~/.ssh/authorized_keys` on all hosts accessible from the runner against a known-good backup or Git-tracked baseline; (4) For 30-day pipeline monitoring, implement a simple SHA-256 checksum verification script that compares `Gemfile.lock` and `go.sum` hashes against a Git-committed baseline on every CI run — alert on any deviation via a pre-build step that calls `sha256sum -c checksums.txt || exit 1`.

Evidence: During recovery validation, collect and retain: (1) AWS CloudTrail event history export covering the full exposure window filtered for IAM write operations (`CreateUser`, `CreateAccessKey`, `AttachUserPolicy`, `CreateRole`, `PutRolePolicy`) — these reveal whether stolen AWS credentials were used to establish persistence; (2) GitHub audit log export for the exposure window showing repository collaborator additions, webhook creations, OAuth app authorizations, and Actions secret modifications that may represent attacker persistence; (3) SSH `AuthorizedKeysFile` snapshots from all hosts reachable from the CI runner, with `stat` timestamps confirming last-write dates; (4) Artifact registry checksums (ECR image digests, RubyGems gemspec hashes, Go module zip hashes) for all packages published or pushed during the exposure window to detect supply-chain poisoning of downstream artifacts.

Step 5: Post-Incident — Evaluate current controls against CWE-494 and CWE-829: implement or enforce dependency pinning with hash verification (e.g., Gemfile.lock with checksum enforcement, go.sum verification). Adopt a software composition analysis (SCA) tool integrated into the CI pipeline to flag new or unrecognized package publishers. Review the principle of least privilege for CI runner credential scopes — runners should not have access to production AWS credentials or broad GitHub token scopes. Map control gaps to NIST SP 800-161 (supply chain risk management) and NIST CSF Respond/Recover functions.

NIST Phase: Post Incident

Reference: NIST 800-61r3 §4 — Post-Incident Activity: conduct lessons-learned review mapping control gaps exposed by the BufferZoneCorp campaign to CWE-494 and CWE-829, enforce dependency integrity verification in CI pipelines, and reduce CI runner credential blast radius through least-privilege scoping.

Controls: NIST SI-2 (Flaw Remediation) — enforce dependency pinning with hash verification in Gemfile.lock and go.sum as a systematic flaw in dependency integrity controls exposed by this campaign, NIST SI-7 (Software, Firmware, and Information Integrity) — implement SCA tooling integrated into CI pipelines to detect unrecognized or newly introduced package publishers such as BufferZoneCorp, NIST SA-12 (Supply Chain Protection) — map control gaps against NIST SP 800-161 supply chain risk management practices, specifically third-party package publisher vetting, NIST AC-6 (Least Privilege) — restrict CI runner IAM roles and GitHub token scopes to minimum required permissions, eliminating production credential access from build environments, NIST IR-8 (Incident Response Plan) — update IR plan to include supply chain compromise scenarios with specific detection criteria for malicious package publishers, CIS 7.1 (Establish and Maintain a Vulnerability Management Process) — incorporate SCA scanning for new or unrecognized package publishers into the vulnerability management process, CIS 7.2 (Establish and Maintain a Remediation Process) — document supply chain compromise remediation procedures, including dependency manifest review and credential rotation workflows, as a repeatable process

Compensating: For teams without commercial SCA tooling: (1) Integrate `bundler-audit` as a mandatory CI step (`gem install bundler-audit && bundle-audit check --update`) to flag gems against the Ruby Advisory Database; (2) Add `govulncheck ./...` as a required Go CI step to detect known-vulnerable module versions; (3) Enforce `go.sum` verification by setting `GONOSUMCHECK=` (empty) and `GOFLAGS=-mod=readonly` in CI to prevent unapproved module additions; (4) Use a free OSSF Scorecard GitHub Action (`ossf/scorecard-action`) to flag new dependency publishers with low trust scores; (5) Scope CI runner IAM roles using AWS IAM Access Analyzer to generate a least-privilege policy from CloudTrail — `aws accessanalyzer start-policy-generation --policy-generation-details 'principalArn=arn:aws:iam:::role/'` — then replace the existing broad role policy with the generated minimal policy.

Evidence: For post-incident documentation, preserve and archive: (1) The complete list of BufferZoneCorp packages identified across RubyGems and Go module ecosystems with their exact version numbers and publication timestamps, sourced from RubyGems.org and pkg.go.dev historical records, as the authoritative scope definition for this campaign; (2) A before/after comparison of CI runner IAM role policies and GitHub token scopes to document the credential blast radius that existed at time of compromise versus the hardened post-incident state; (3) Dependency manifest diffs (Gemfile.lock and go.sum before and after eradication) showing the exact packages removed; (4) SCA scan results from rebuilt environments confirming absence of BufferZoneCorp packages and establishing a verified clean baseline for future comparison.

Detection Guidance

Primary detection focus: package installation events and anomalous outbound network activity from CI runners. (1) Audit package manager logs for any gem or Go module installed from the 'BufferZoneCorp' namespace; cross-reference Gemfile.lock and go.sum against known-good states using version control history. Specific package names: [confirm and list if available]. (2) Query SIEM or CI platform logs for outbound HTTP/S connections initiated by build runner processes to IP addresses or domains not in an allowlist, particularly during or immediately after dependency installation steps. (3) Check for unexpected modifications to credential files: ~/.aws/credentials, ~/.ssh/authorized_keys, ~/.npmrc, and GitHub CLI config (~/.config/gh/). File integrity monitoring alerts on these paths are a direct indicator. (4) Review GitHub Actions audit logs (available under Organization Settings > Audit log) for unexpected secret access events or workflow modifications coinciding with the exposure window. (5) Behavioral indicator: delayed-activation packages may communicate externally on a trigger event; monitor for periodic beacon-like outbound connections from build infrastructure to unknown endpoints even after the initial installation window has passed (T1053, T1071.001 pattern). IOC availability (package hashes, command-and-control IPs) depends on threat intelligence vendor reporting and may not be present in public news coverage. Organizations should request indicators from CISA, vendor threat intelligence platforms, or the RubyGems and Go security teams. Treat absence of confirmed public IOCs as a sourcing gap, not a clearance.

Indicators of Compromise

Type	Value	Context	Confidence
DOMAIN	`github.com/BufferZoneCorp`	GitHub account used to publish malicious Ruby gems and Go modules in this campaign	HIGH

Framework Mappings

MITRE-ATTACK

- **T1053** — Scheduled Task/Job
- **T1059** — Command and Scripting Interpreter
- **T1552.001** — Credentials In Files
- **T1098.004** — SSH Authorized Keys
- **T1547** — Boot or Logon Autostart Execution

- **T1071.001** — Web Protocols
- **T1608.001** — Upload Malware
- **T1552.004** — Private Keys
- **T1528** — Steal Application Access Token
- **T1554** — Compromise Host Software Binary
- **T1195.001** — Compromise Software Dependencies and Development Tools

NIST-800-53R5

- **AC-3** — Access Enforcement
- **CM-7** — Least Functionality
- **SI-3** — Malicious Code Protection
- **SI-4** — System Monitoring
- **SI-7** — Software, Firmware, and Information Integrity
- **CM-3** — Configuration Change Control
- **SR-2** — Supply Chain Risk Management Plan

OWASP-TOP10-2021

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

CIS-V8

- **2.5** — Allowlist Authorized Software
- **2.6** — Allowlist Authorized Libraries
- **6.3** — Require MFA for Externally-Exposed Applications
- **15.1** — Establish and Maintain an Inventory of Service Providers

HIPAA-SECURITY

- **164.312(d)** — Person or Entity Authentication

SOC2-TSC

- **CC6.1** — Logical access security software, infrastructure, and architectures
- **CC9.2** — Manages risks associated with vendors and business partners

NIST-CSF-2

- **GV.SC-01** — Cybersecurity supply chain risk management program

ISO-27001-2022

- **A.5.21** — Managing information security in the ICT supply chain
- **A.5.23** — Information security for use of cloud services

MITRE ATT&CK Mapping

Technique ID	Technique Name	Tactic
T1053	Scheduled Task/Job	Execution
T1059	Command and Scripting Interpreter	Execution
T1552.001	Credentials In Files	Credential-Access
T1098.004	SSH Authorized Keys	Persistence
T1547	Boot or Logon Autostart Execution	Persistence
T1071.001	Web Protocols	Command-And-Control
T1608.001	Upload Malware	Resource-Development
T1552.004	Private Keys	Credential-Access
T1528	Steal Application Access Token	Credential-Access
T1554	Compromise Host Software Binary	Persistence
T1195.001	Compromise Software Dependencies and Development Tools	Initial-Access

Sources

Source	URL	Tier
Security News	https://thehackernews.com/2026/05/poisoned-ruby-gems-and-go-modules.. .	T3
Poisoned Ruby gems + Go modules used in a supply chain attack ...	https://www.facebook.com/thehackernews/posts/-poisoned-ruby-gems-go...	T3
Malicious Ruby Gems Go Modules CI Secret Theft Explained	https://thecybrdef.com/malicious-ruby-gems-go-modules-steal-dev-sec...	T3
Malicious Ruby Gems and Go Modules Impersonate Developer...	https://app.daily.dev/posts/malicious-ruby-gems-and-go-modules-impe...	T3
A wave of attacks is using layered npm dependencies to ... - Facebook	https://www.facebook.com/thehackernews/posts/-a-wave-of-attacks-is-...	T3

Generated 2026-05-01 18:54 UTC by TJS Security Command Center