INTELLIGENCE BRIEFING
Security Command Center
TLP:CLEAR
2026-04-30 06:30 UTC

SAP npm Packages Weaponized in TeamPCP Supply-Chain Operation Targeting Enterprise CI/CD Pipelines

THREAT CAMPAIGN | **CRITICAL** | CVSS 9.5

SCC Item ID	SCC-CAM-2026-0242
Type	Threat Campaign
Severity	CRITICAL
CVSS Base Score	9.5
Affected Products	SAP npm packages: @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, mbt v1.2.48; SAP Cloud Application Programming Model (CAP); SAP Cloud MTA; CI/CD environments (CircleCI, GitHub Actions); AWS, Azure, Google Cloud credential stores; Kubernetes clusters
Published	2026-04-29T18:43:44
Discovery Source	Rss

Executive Summary

Four official SAP npm packages used in enterprise cloud application development were compromised with credential-harvesting code that executes automatically during routine software builds. Any organization that has installed @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, or mbt v1.2.48 is at risk of having cloud credentials, SSH keys, Kubernetes secrets, and CI/CD pipeline tokens silently stolen. The attack self-propagates by using stolen npm credentials to compromise additional packages, meaning the blast radius extends beyond the initial four packages.

Technical Analysis

Four SAP npm packages were trojanized with a malicious preinstall script that executes on npm install: @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, and mbt v1.2.48. The preinstall hook targets cloud provider credential files (AWS ~/.aws/credentials, Azure service principal tokens, GCP application default credentials), SSH private keys, Kubernetes secrets (including in-cluster service account tokens), and CI/CD environment variables and tokens (CircleCI, GitHub Actions). Attribution is pending authoritative confirmation from SAP or CISA. Probable initial access vector: exposed npm publish token leaked via a misconfigured CircleCI job, consistent with CWE-732 (incorrect permission assignment for critical resource). The malware exfiltrates harvested material over network channels (T1041) and leverages stolen npm

credentials to trojanize additional packages (T1195.001, T1195.002), creating a self-amplifying supply-chain footprint. Relevant CWEs: CWE-312 (cleartext storage of sensitive information), CWE-522 (insufficiently protected credentials), CWE-732 (incorrect permission assignment), CWE-506 (embedded malicious code). No CVE assigned as of this report. No vendor CVSS vector available. Priority score: 0.85. Source quality is T2/T3 (moderate, 0.48); authoritative confirmation from SAP Security Advisory or CISA is pending. Organizations should apply enhanced scrutiny to detections and correlate findings with official SAP advisories before implementing containment actions.

Action Checklist

1. Step 1: Containment. Immediately block or quarantine any build environment that has installed @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, or mbt v1.2.48. Suspend all npm publish operations using tokens stored in affected CI/CD pipelines. Rotate all cloud provider credentials (AWS IAM keys, Azure service principals, GCP service account keys), SSH keys, and Kubernetes service account tokens on any system where these packages were installed.
2. Step 2: Detection. Query npm audit logs and package-lock.json / yarn.lock files across all repositories for the four affected package names and exact version strings. Search CI/CD pipeline logs (CircleCI build output, GitHub Actions workflow logs) for unexpected outbound network connections or process spawns during npm install steps. On developer workstations, check ~/.npm/_logs for preinstall script execution. Review cloud provider access logs (AWS CloudTrail, Azure Monitor, GCP Audit Logs) for credential use from unexpected source IPs or at unusual times coinciding with build activity. Audit CI/CD secret manager audit logs (e.g., HashiCorp Vault, AWS Secrets Manager) for npm token creation or rotation events during the timeframe when affected packages were in use.
3. Step 3: Eradication. Downgrade or remove the four compromised packages and replace with the last known-clean versions, verified against the official SAP cap-js/cds-dbs GitHub repository commit history and cross-referenced with any commit GPG signatures or checksums published by SAP. Remove any npm publish tokens associated with affected pipelines and issue new tokens with least-privilege scopes. Audit all packages published by the affected npm accounts for secondary compromise, as the self-propagation mechanism may have trojanized downstream dependencies.
4. Step 4: Recovery. After credential rotation and isolation of compromised systems, validate that no unauthorized cloud resources were provisioned (check for new IAM roles, EC2/VM instances, storage buckets, or Kubernetes workloads). Re-run builds using clean package versions in isolated environments before restoring production pipelines. Enable npm package integrity verification (npm ci with lockfile enforcement) and confirm builds complete without unexpected preinstall script output. Monitor cloud provider billing dashboards for anomalous resource consumption indicative of ongoing unauthorized access.
5. Step 5: Post-Incident. Audit all CI/CD pipeline configurations for exposed publish tokens or secrets in environment variables accessible to build logs. Implement npm publish token scoping to restrict tokens to specific package scopes. Evaluate adoption of a private npm registry or artifact proxy (e.g., Artifactory, Nexus) to gate third-party package ingestion. Map this incident to MITRE T1195.001 and T1195.002 and assess whether existing supply-chain controls (dependency review, SBOM generation, preinstall script blocking via --ignore-scripts) are enforced consistently across all build pipelines.

IR / Forensic Enrichment

Triage Priority	IMMEDIATE
Escalation Criteria	Escalate to CISO, Legal, and external IR retainer immediately if AWS CloudTrail, Azure Monitor, or GCP Audit Logs confirm any API call using a credential present in an affected build environment originated from a non-corporate IP, or if any Kubernetes secret, database connection string, or cloud service account key from an affected pipeline is confirmed to have been accessed post-build — these conditions indicate active attacker use of exfiltrated credentials and may trigger breach notification obligations under GDPR, CCPA, or applicable cloud service agreements.
Recovery Notes	After rotating all credentials, maintain continuous CloudTrail/Azure Monitor/GCP Audit Log alerting on the rotated key IDs for a minimum of 30 days to detect any delayed use of credentials exfiltrated before rotation — attackers in supply-chain operations frequently stage stolen credentials for weeks before operational use. Re-run all production builds in an isolated environment using 'npm ci --ignore-scripts' with the clean package versions pinned in an updated lockfile, and compare the full dependency tree SHA against the pre-incident baseline before promoting to production. Given the self-propagation mechanism's potential to trojanize additional SAP CAP ecosystem packages, subscribe to npm security advisories for the entire @cap-js and @sap scope and configure Dependabot or Renovate Bot alerts specifically for unexpected version changes in mbt, @cap-js/sqlite, @cap-js/postgres, and @cap-js/db-service for a minimum of 90 days post-incident.
Forensic Artifacts	~/.npm/_logs/*.log on CI/CD runners and developer workstations — these npm debug logs record the exact timestamp and exit code of every preinstall script execution, providing definitive evidence of whether the malicious scripts in @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, or mbt v1.2.48 fired on a given system | node_modules/@cap-js/sqlite/package.json, node_modules/@cap-js/db-service/package.json, and equivalent paths for all four compromised packages — the 'scripts.preinstall' field and any referenced script files contain the actual credential-harvesting payload and must be preserved with cryptographic hashes before node_modules is deleted | AWS CloudTrail logs, Azure Monitor Activity Logs, and GCP Audit Logs filtered by the exact IAM key IDs, service principal client IDs, and GCP service account emails that were present as environment variables in affected build jobs — these logs constitute the primary evidence of whether exfiltrated credentials were operationalized for unauthorized cloud resource access | CI/CD pipeline build logs (GitHub Actions workflow run logs and CircleCI build output) for every job executing 'npm install' or 'npm ci' during the 90-day window prior to detection — specifically the stdout/stderr captured during the install phase, which will contain any output, error messages, or network activity generated by the malicious preinstall scripts in the four TeamPCP-compromised packages | Kubernetes audit logs filtered for secret read events (verb: 'get', resource: 'secrets') and any new RoleBinding or ClusterRoleBinding objects created during the compromise window — the TeamPCP campaign targeted Kubernetes secrets specifically, making these logs the definitive evidence of whether cluster credentials were accessed or attacker persistence was established via new role assignments

Per-Action IR Details

Step 1: Containment — Immediately block or quarantine any build environment that has installed @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, or mbt v1.2.48. Suspend all npm publish operations using tokens stored in affected CI/CD pipelines. Rotate all cloud provider credentials (AWS IAM keys, Azure service principals, GCP service account keys), SSH keys, and Kubernetes service account tokens on any system where these packages were installed.

NIST Phase: Containment

Reference: NIST 800-61r3 §3.3 — Containment Strategy: isolate affected systems to prevent further credential exfiltration while preserving forensic state before eradication begins.

Controls: NIST IR-4 (Incident Handling), NIST AC-2 (Account Management) — revoke and rotate compromised CI/CD service identities, NIST SC-28 (Protection of Information at Rest) — Kubernetes secrets and cloud credentials at rest were exposed by preinstall script execution, CIS 5.3 (Disable Dormant Accounts) — immediately disable any npm publish tokens associated with the four compromised package scopes, CIS 6.2 (Establish an Access Revoking Process) — systematically revoke AWS IAM keys, Azure service principals, GCP service account keys, and Kubernetes service account tokens confirmed present in affected build environments

Compensating: On Linux/macOS CI runners, immediately kill and disable the build agent service: 'sudo systemctl stop circleci-agent' or cancel active GitHub Actions workflows via the GitHub API ('gh run cancel '). For AWS, use the CLI to deactivate all IAM access keys found in build environment variables: 'aws iam update-access-key --access-key-id --status Inactive --user-name '. For Kubernetes, enumerate and delete compromised service account tokens: 'kubectl get secrets --all-namespaces | grep kubernetes.io/service-account-token', then 'kubectl delete secret -n '. Snapshot the runner filesystem with 'dd' or 'tar czf /tmp/runner_snapshot.tar.gz /home/runner' before any remediation to preserve forensic evidence.

Evidence: BEFORE rotating credentials, capture: (1) the exact contents of ~/.npm/_logs/ on each affected runner showing preinstall script invocation for @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, or mbt v1.2.48; (2) a full dump of all environment variables present during the build ('printenv > /tmp/env_capture.txt' on Linux runners) to establish which secrets were in scope at execution time; (3) network connection state at time of containment ('ss -tulnp' or 'netstat -anp') to identify any active exfiltration channels opened by the malicious preinstall scripts; (4) the CI/CD pipeline's secret store configuration showing which cloud credential environment variables (e.g., AWS_ACCESS_KEY_ID, AZURE_CLIENT_SECRET, GOOGLE_APPLICATION_CREDENTIALS) were injected into the compromised build job.

Step 2: Detection — Query npm audit logs and package-lock.json / yarn.lock files across all repositories for the four affected package names and exact version strings. Search CI/CD pipeline logs (CircleCI build output, GitHub Actions workflow logs) for unexpected outbound network connections or process spawns during npm install steps. On developer workstations, check ~/.npm/_logs for preinstall script execution. Review cloud provider access logs (AWS CloudTrail, Azure Monitor, GCP Audit Logs) for credential use from unexpected source IPs or at unusual times coinciding with build activity. Check for new or modified npm publish tokens in CI/CD secret stores.

NIST Phase: Detection Analysis

Reference: NIST 800-61r3 §3.2 — Detection and Analysis: correlate build-time artifact evidence with cloud provider access logs to establish whether credential exfiltration resulted in unauthorized cloud API activity.

Controls: NIST IR-5 (Incident Monitoring) — track and document each repository and pipeline confirmed to have resolved the four compromised package versions, NIST AU-2 (Event Logging) — CI/CD pipeline logs and cloud provider audit trails are the primary log sources for detecting exfiltration from npm preinstall script execution, NIST AU-6 (Audit Record Review, Analysis, and Reporting) — correlate npm install timestamps from build logs against AWS CloudTrail, Azure Monitor Activity Logs, and GCP Audit Logs for API calls from anomalous IPs, NIST SI-4 (System Monitoring) — monitor for outbound connections from build runners to attacker-controlled exfiltration endpoints initiated during 'npm install' execution, CIS 8.2 (Collect Audit Logs) — ensure CircleCI build output logs and GitHub Actions workflow logs are centrally retained before investigating

Compensating: Use this one-liner to recursively find all package-lock.json files and grep for the four compromised versions: 'find /path/to/repos -name "package-lock.json" -exec grep -l -E "@cap-js/sqlite.*2\.2\.2|@cap-js/postgres.*2\.2\.2|@cap-js/db-service.*2\.10\.1|mbt.*1\.2\.48" {} \;'. For GitHub Actions, query workflow run logs via CLI: 'gh run list --limit 100 --json databaseId,conclusion,createdAt | jq .[].databaseId | xargs -I{} gh run view {} --log | grep -E "(preinstall|postinstall|curl|wget|nc|bash -i|python -c)"'. For AWS CloudTrail without a SIEM, download logs and parse with jq: 'cat *.json | jq ".Records[] | select(.userIdentity.accessKeyId==\"\")"' to identify all API calls made with the exfiltrated key. On developer workstations, check npm debug logs: 'cat ~/.npm/_logs/*.log | grep -E "preinstall|run script"'.

Evidence: Capture before analysis is complete: (1) raw GitHub Actions workflow log files (.txt) and CircleCI build artifacts for every job that executed 'npm install' or 'npm ci' within the past 90 days — specifically the stdout/stderr output surrounding the install step where preinstall scripts for @cap-js/db-service or mbt would have fired; (2) AWS CloudTrail logs filtered for the time window of each affected build job, looking for AssumeRole, GetSecretValue, ListBuckets, or DescribeInstances API calls from source IPs not matching known corporate egress ranges; (3) DNS query logs from the build runner (if available via VPC Flow Logs or resolver logging) showing outbound DNS lookups to attacker-controlled domains initiated by the npm preinstall script; (4) the package-lock.json or yarn.lock file at the exact Git commit SHA used in each compromised build, preserved as immutable evidence of which dependency tree was resolved.

Step 3: Eradication — Downgrade or remove the four compromised packages and replace with the last known-clean versions (verify against the official SAP cap-js/cds-dbs GitHub repository commit history for clean version tags). Remove any npm publish tokens associated with affected pipelines and issue new tokens with least-privilege scopes. Audit all packages published by the affected npm accounts for secondary compromise, as the self-propagation mechanism may have trojanized downstream dependencies.

NIST Phase: Eradication

Reference: NIST 800-61r3 §3.4 — Eradication: remove all instances of the compromised package versions from the dependency tree and revoke the npm publish credentials used by the self-propagation mechanism before any package re-publication occurs.

Controls: NIST SI-2 (Flaw Remediation) — pin all four packages to verified clean versions from the SAP cap-js/cds-dbs GitHub repository and enforce lockfile integrity going forward, NIST CM-2 (Baseline Configuration) — update the approved software baseline to exclude @cap-js/sqlite v2.2.2, @cap-js/postgres v2.2.2, @cap-js/db-service v2.10.1, and mbt v1.2.48 explicitly, NIST IR-4 (Incident Handling) — the self-propagation mechanism that uses stolen npm tokens to trojanize downstream packages requires treating each newly identified compromised package as a sub-incident requiring its own eradication track, CIS 2.2 (Ensure Authorized Software is Currently Supported) — flag the four compromised versions as unauthorized and remove from all build environments, CIS 7.2 (Establish and Maintain a Remediation Process) — document the clean version pins and verification steps against SAP GitHub commit history as the remediation baseline for this incident

Compensating: Verify clean version integrity by checking the npm package tarball hash against the corresponding Git tag on the official SAP repository before pinning: 'npm pack @cap-js/sqlite@ && shasum -a 256 cap-js-sqlite-.tgz' then compare to the hash of the tarball built from the tagged source at 'https://github.com/cap-js/cds-dbs'. To identify all packages published by the compromised npm accounts (which the self-propagation mechanism may have touched), query the npm registry: 'npm search --json maintainer:' and cross-reference publication timestamps against the known compromise window. Use YARA to scan build caches and node_modules directories for the credential-harvesting payload pattern before eradication: write a rule matching strings from the malicious preinstall script (e.g., environment variable enumeration patterns, curl/wget exfiltration calls) and run 'yara -r malicious_preinstall.yar ./node_modules'.

Evidence: Before removing node_modules: (1) preserve a complete copy of the node_modules directory from each affected build environment ('tar czf node_modules_evidence__.tar.gz ./node_modules') to enable offline analysis of the exact malicious preinstall script code that executed; (2) extract and hash the specific preinstall script from the compromised packages: 'cat node_modules/@cap-js/sqlite/package.json | jq .scripts.preinstall' and 'shasum -a 256 node_modules/@cap-js/sqlite/preinstall.js' (or equivalent script file) to establish artifact hashes for IOC sharing; (3) document the full dependency tree at time of compromise: 'npm list --all --json > dependency_tree_evidence.json' to identify all packages that transitively depended on the four compromised packages, which may have been re-published with the self-propagation payload.

Step 4: Recovery — After credential rotation, validate that no unauthorized cloud resources were provisioned (check for new IAM roles, EC2/VM instances, storage buckets, or Kubernetes workloads). Re-run builds using clean package versions in isolated environments before restoring production pipelines. Enable npm package integrity verification (npm ci with lockfile enforcement) and confirm builds complete without unexpected preinstall script output. Monitor cloud provider billing dashboards for anomalous resource consumption indicative of ongoing unauthorized access.

NIST Phase: Recovery

Reference: NIST 800-61r3 §3.5 — Recovery: validate that all exfiltrated credentials have been fully invalidated and that no attacker-established cloud persistence (IAM backdoors, unauthorized workloads) survives into the restored environment before production pipelines are brought back online.

Controls: NIST IR-4 (Incident Handling) — recovery actions must include verification that the self-propagation vector (stolen npm publish tokens) cannot re-introduce compromised packages into restored pipelines, NIST SI-7 (Software, Firmware, and Information Integrity) — enforce npm ci with lockfile integrity verification on all restored pipelines to detect any future tampering with @cap-js or mbt package versions, NIST CM-6 (Configuration Settings) — enforce '--ignore-scripts' flag or equivalent preinstall script blocking as a pipeline configuration baseline before restoring production builds, CIS 1.1 (Establish and Maintain Detailed Enterprise Asset Inventory) — enumerate all cloud resources (IAM roles, EC2 instances, storage buckets, Kubernetes workloads) created after the earliest confirmed compromise timestamp to identify unauthorized provisioning, CIS 7.3 (Perform Automated Operating System Patch Management) — re-run builds only after confirming all four compromised package versions are removed from both npm cache and node_modules across all pipeline runners

Compensating: For AWS, enumerate all IAM entities created after the earliest confirmed compromise date: 'aws iam get-credential-report' (generate first with 'aws iam generate-credential-report'), then filter for creation dates overlapping the incident window. Check for unauthorized EC2 instances: 'aws ec2 describe-instances --query "Reservations[*].Instances[*].[InstanceId,LaunchTime,Tags]" --output table | grep -v '. For Kubernetes, list all workloads created in the compromise window: 'kubectl get all --all-namespaces -o json | jq ".items[] | select(.metadata.creationTimestamp > \"\")"'. For build integrity verification, configure npm ci with: 'npm ci --ignore-scripts' and capture full stdout/stderr to confirm no preinstall script output appears for any @cap-js or mbt package. Monitor AWS billing anomalies via: 'aws ce get-cost-and-usage --time-period Start=,End= --granularity DAILY --metrics BlendedCost'.

Evidence: Before declaring recovery complete, collect: (1) a final AWS CloudTrail export, Azure Activity Log export, and GCP Audit Log export covering the full period from earliest confirmed package installation to present, to establish a complete timeline of all API activity using the now-rotated credentials — this is the definitive record of attacker actions in cloud environments; (2) Kubernetes audit logs ('kubectl logs -n kube-system kube-apiserver-') filtered for service account tokens that have since been rotated, to identify any unauthorized API calls (pod creation, secret reads, role bindings) made during the compromise window; (3) a clean build log from the first successful 'npm ci --ignore-scripts' run on the restored pipeline, serving as the verified baseline for future integrity comparisons.

Step 5: Post-Incident — Audit all CI/CD pipeline configurations for exposed publish tokens or secrets in environment variables accessible to build logs. Implement npm publish token scoping to restrict tokens to specific package scopes. Evaluate adoption of a private npm registry or artifact proxy (e.g., Artifactory, Nexus) to gate third-party package ingestion. Map this incident to MITRE T1195.001 and T1195.002 and assess whether existing supply-chain controls (dependency review, SBOM generation, preinstall script blocking via --ignore-scripts) are enforced consistently across all build pipelines.

NIST Phase: Post Incident

Reference: NIST 800-61r3 §4 — Post-Incident Activity: document lessons learned specific to npm supply-chain trust assumptions, update detection rules to flag preinstall script execution in build environments, and share IOCs (compromised package hashes, exfiltration endpoints) with relevant ISACs or CISA.

Controls: NIST IR-4 (Incident Handling) — update the incident handling playbook to add supply-chain package compromise as a named incident category with the specific triage steps exercised in this incident, NIST IR-8 (Incident Response Plan) — revise the IR plan to include npm registry monitoring and SBOM-based dependency alerting as standing preparation controls following the TeamPCP campaign, NIST SI-2 (Flaw Remediation) — establish a formal process for monitoring SAP CAP ecosystem packages (cap-js, cds-dbs, mbt) for new malicious versions, including npm package integrity alerts, NIST RA-3 (Risk Assessment) — re-assess supply-chain risk for all CI/CD pipelines that consume open-source npm packages, specifically documenting the risk of preinstall script execution with access to injected secrets, CIS 7.1 (Establish and Maintain a Vulnerability Management Process) — add npm package integrity verification and SBOM generation to the vulnerability management process scope, CIS 2.3 (Address Unauthorized Software) — enforce allowlisting of approved npm package versions in all pipelines; treat any @cap-js or mbt version outside the approved baseline as unauthorized software requiring immediate review

Compensating: Generate a retroactive SBOM for all affected repositories using Syft (free, open-source): 'syft dir:. -o cyclonedx-json > sbom__.json' and diff against a pre-incident baseline to identify all packages introduced during the compromise window. Block preinstall scripts globally in all pipelines by adding '.npmrc' to each repository root with 'ignore-scripts=true' and enforcing it via a GitHub Actions required workflow or CircleCI orb pre-step. For Sigma-based detection, write a rule matching process creation events where the parent process is 'npm' or 'node' and child processes include 'curl', 'wget', 'bash -c', or 'python -c' during CI build jobs — this detects the preinstall script execution pattern used by the TeamPCP campaign. Submit the compromised package hashes and any identified exfiltration domain/IP IOCs to CISA's automated indicator sharing (AIS) program and the npm security team at 'security@npmjs.com' to trigger package removal and downstream notifications.

Evidence: For the lessons-learned record and threat intelligence sharing: (1) the extracted and hashed malicious preinstall scripts from all four compromised packages, with deobfuscated analysis documenting exactly which environment variables, file paths (e.g., ~/.aws/credentials, ~/.kube/config, ~/.ssh/id_rsa), and credential patterns the TeamPCP campaign targeted; (2) a complete list of all repository names, pipeline IDs, build timestamps, and runner hostnames that resolved the four compromised package versions, serving as the definitive blast-radius document for regulatory notification assessment; (3) any attacker-controlled exfiltration endpoints (domains, IPs) identified from DNS logs or network captures during Step 2, formatted as STIX 2.1 indicators for sharing with sector ISACs and CISA under the MITRE T1195.001 (Compromise Software Dependencies and Development Tools) and T1195.002 (Compromise Software Supply Chain) technique mappings.

Detection Guidance

Primary detection targets: (1) Package presence, scan all package-lock.json, yarn.lock, and npm-shrinkwrap.json files for @cap-js/sqlite@2.2.2, @cap-js/postgres@2.2.2, @cap-js/db-service@2.10.1, mbt@1.2.48. (2) Preinstall script execution, in CI/CD logs, search for script execution events during npm install phases; flag any child process spawned by node that makes outbound network connections. (3) Credential file access, on Linux/macOS workstations and build agents, monitor file access events (auditd or EDR telemetry) on ~/.aws/credentials, ~/.config/gcloud/, ~/.kube/config, ~/.ssh/id_* during or shortly after npm install. (4) Anomalous outbound traffic, look for DNS queries or TCP connections to uncommon destinations originating from build agent processes during dependency installation windows. (5) npm token misuse, contact npm support to request account audit logs for CI/CD service accounts, or monitor npm package version history on npmjs.com for unexpected version publications to owned packages. Cross-reference timestamps with CI/CD pipeline execution logs. (6) Cloud credential anomalies, correlate AWS CloudTrail / Azure Activity Log / GCP Audit Log for API calls using keys that were present in affected build environments, filtering for source IPs inconsistent with known build agent ranges. No confirmed IOC hashes or C2 infrastructure are available at current source quality (0.48); update detection rules as authoritative indicators are published by SAP or CISA.

Framework Mappings

MITRE-ATTACK

- **T1102** — Web Service
- **T1003** — OS Credential Dumping
- **T1552.004** — Private Keys
- **T1195.001** — Compromise Software Dependencies and Development Tools
- **T1059.007** — JavaScript
- **T1059** — Command and Scripting Interpreter
- **T1528** — Steal Application Access Token

- **T1552.001** — Credentials In Files
- **T1041** — Exfiltration Over C2 Channel
- **T1552.007** — Container API
- **T1176** — Software Extensions
- **T1078.004** — Cloud Accounts
- **T1027** — Obfuscated Files or Information
- **T1195.002** — Compromise Software Supply Chain
- **T1057** — Process Discovery

NIST-800-53R5

- **AC-6** — Least Privilege
- **IA-5** — Authenticator Management
- **SI-4** — System Monitoring
- **CM-7** — Least Functionality
- **SI-3** — Malicious Code Protection
- **SI-7** — Software, Firmware, and Information Integrity
- **CA-7** — Continuous Monitoring
- **SC-7** — Boundary Protection
- **SA-9** — External System Services
- **SR-3** — Supply Chain Controls and Processes
- **AC-3** — Access Enforcement

OWASP-TOP10-2021

- **A04:2021** — Insecure Design
- **A07:2021** — Identification and Authentication Failures
- **A01:2021** — Broken Access Control

CIS-V8

- **5.2** — Use Unique Passwords
- **3.3** — Configure Data Access Control Lists
- **6.3** — Require MFA for Externally-Exposed Applications

HIPAA-SECURITY

- **164.308(a)(5)(ii)(D)** — Password Management
- **164.312(d)** — Person or Entity Authentication

SOC2-TSC

- **CC6.1** — Logical access security software, infrastructure, and architectures

ISO-27001-2022

- **A.5.23** — Information security for use of cloud services

MITRE ATT&CK Mapping

Technique ID	Technique Name	Tactic
T1102	Web Service	Command-And-Control
T1003	OS Credential Dumping	Credential-Access
T1552.004	Private Keys	Credential-Access
T1195.001	Compromise Software Dependencies and Development Tools	Initial-Access
T1059.007	JavaScript	Execution
T1059	Command and Scripting Interpreter	Execution
T1528	Steal Application Access Token	Credential-Access
T1552.001	Credentials In Files	Credential-Access
T1041	Exfiltration Over C2 Channel	Exfiltration
T1552.007	Container API	Credential-Access
T1176	Software Extensions	Persistence
T1078.004	Cloud Accounts	Defense-Evasion
T1027	Obfuscated Files or Information	Defense-Evasion
T1195.002	Compromise Software Supply Chain	Initial-Access
T1057	Process Discovery	Discovery

Sources

Source	URL	Tier
Security News	https://www.bleepingcomputer.com/news/security/official-sap-npm-pac...	T3
@cap-js/sqlite - npm	https://www.npmjs.com/package/%40cap-js%2Fsqlite	T3
cap-js/cds-dbs: Monorepo for SQL Database Services for CAP · GitHub	https://github.com/cap-js/cds-dbs	T3
@cap-js/db-service - npm	https://www.npmjs.com/package/%40cap-js%2Fdb-service	T3

Source	URL	Tier
Run a CAP app using SQLite on SAP BTP - Cloud Foundry	https://community.sap.com/t5/technology-blog-posts-by-members/sap-c...	T3

Generated 2026-04-30 06:30 UTC by TJS Security Command Center