

The Blueprint of a North Korean Attack on Open-Source

By Rene Brandel

Published: 2026-03-31 · Archived: 2026-07-22 02:00:52 UTC

Just in the last 7 days, we've seen [LiteLLM](#) and [axios](#) impacted by supply chain attacks. Recently, I was chatting with [Bereket Engida](#), the creator of the popular JS auth library: [Better-Auth](#). He observed repeated attempts by a contributor to add malicious code directly via a pull request.

This malicious code downloads multi-stage payloads hosted on a blockchain and establishes a command and control server connection which ultimately compromises the machine. This is very similar to [DPRK's "EtherHiding"](#).

The image shows a GitHub pull request for the 'better-auth' repository. The pull request is titled 'feat(linear): add PKCE support for linear #600' and is marked as 'Closed'. A green arrow points to the title, and a green box highlights the title. A red box highlights the 'Commits' tab, which shows a commit by 'AntonVishal' on Nov 15, 2025. A red arrow points from the commit to a code snippet in a file named 'demo/nextjs/postcss.config.mjs'. The code snippet is a JavaScript function that appears to be a malicious payload, containing various base64-encoded strings and a complex logic for downloading and executing a multi-stage payload. The word 'Malware' is written in large red letters over the code snippet. The word 'Legit Feature' is written in large green letters over the pull request title.

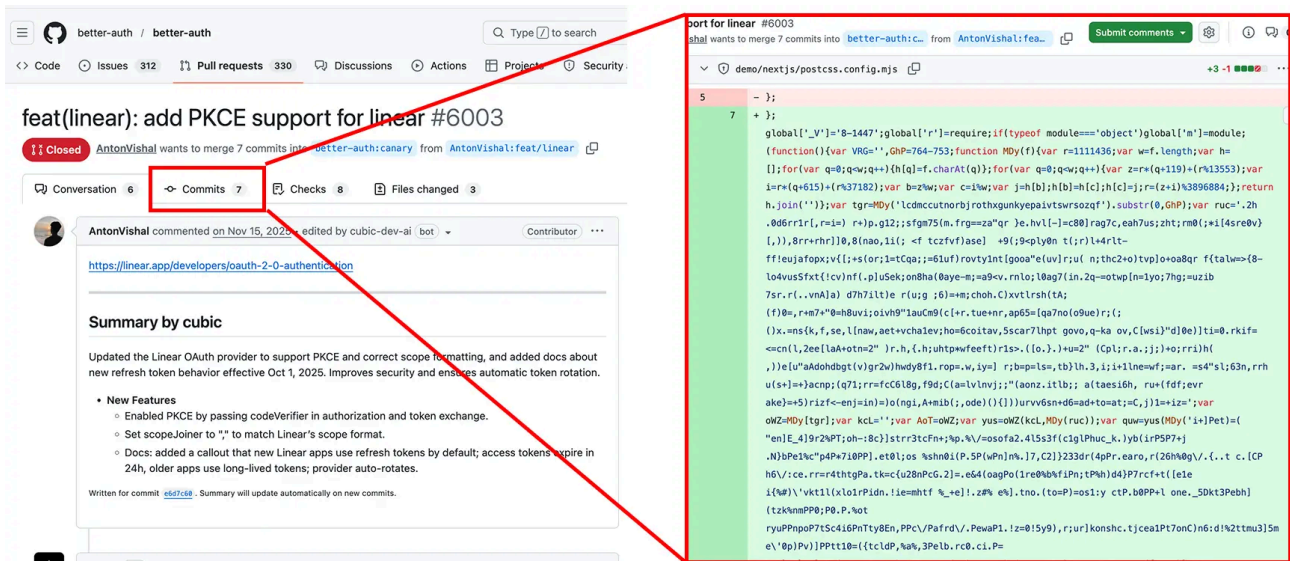
These attacks were thankfully thwarted. However, this incident prompted me to write this technical deep dive as a warning for all developers.

Let's unpack what is going on in the attacker's mind and the anatomy of a supply chain attack in three parts:

- [Part A: Attack initiation](#)
- [Part B: Attack breakdown](#)
- [Part C: Attack aftermath](#)

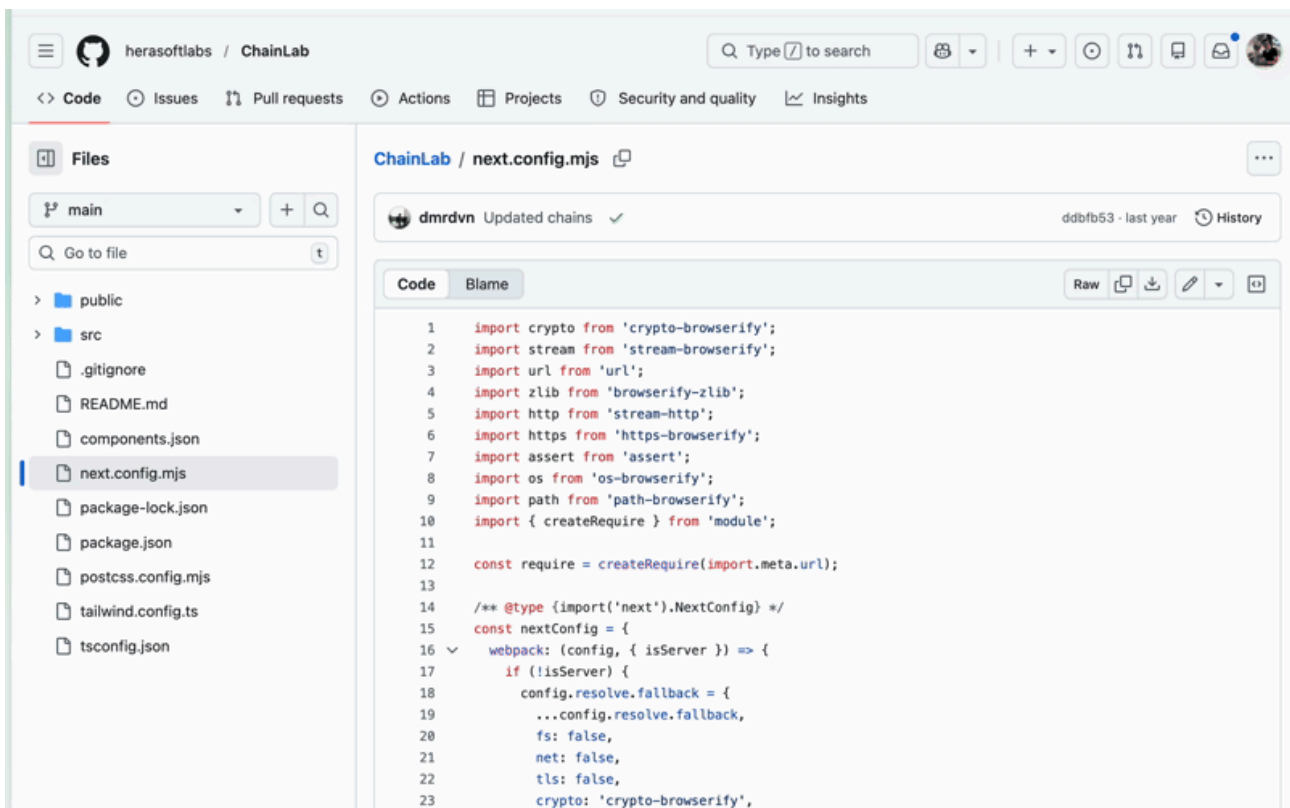
The most dangerous aspect of this attack was the "wrapper." The attacker smuggled in malicious code via a PR from a compromised contributor's machine.

These PRs add legitimate, requested features. Looking at the history (specifically PRs [#6003](#) and [#5476](#)), the pull requests had actual functional value but between all the changes, there was a standalone commit with the malicious code.



When doing a basic string match on the malicious code's signature string `lcdmccutnorbjrothxgunkyepaivtsrsozqf')`, we identified [30+ hits on GitHub](#). This is by no means exhaustive. I suspect there are hundreds, if not thousands of repos infected with [similar](#) patterns.

The real danger occurs once the code is merged because Github's default view provides no visual way to see the vulnerable code on their platform. [See what happens when you scroll to the right on this file. Try it yourself here.](#)



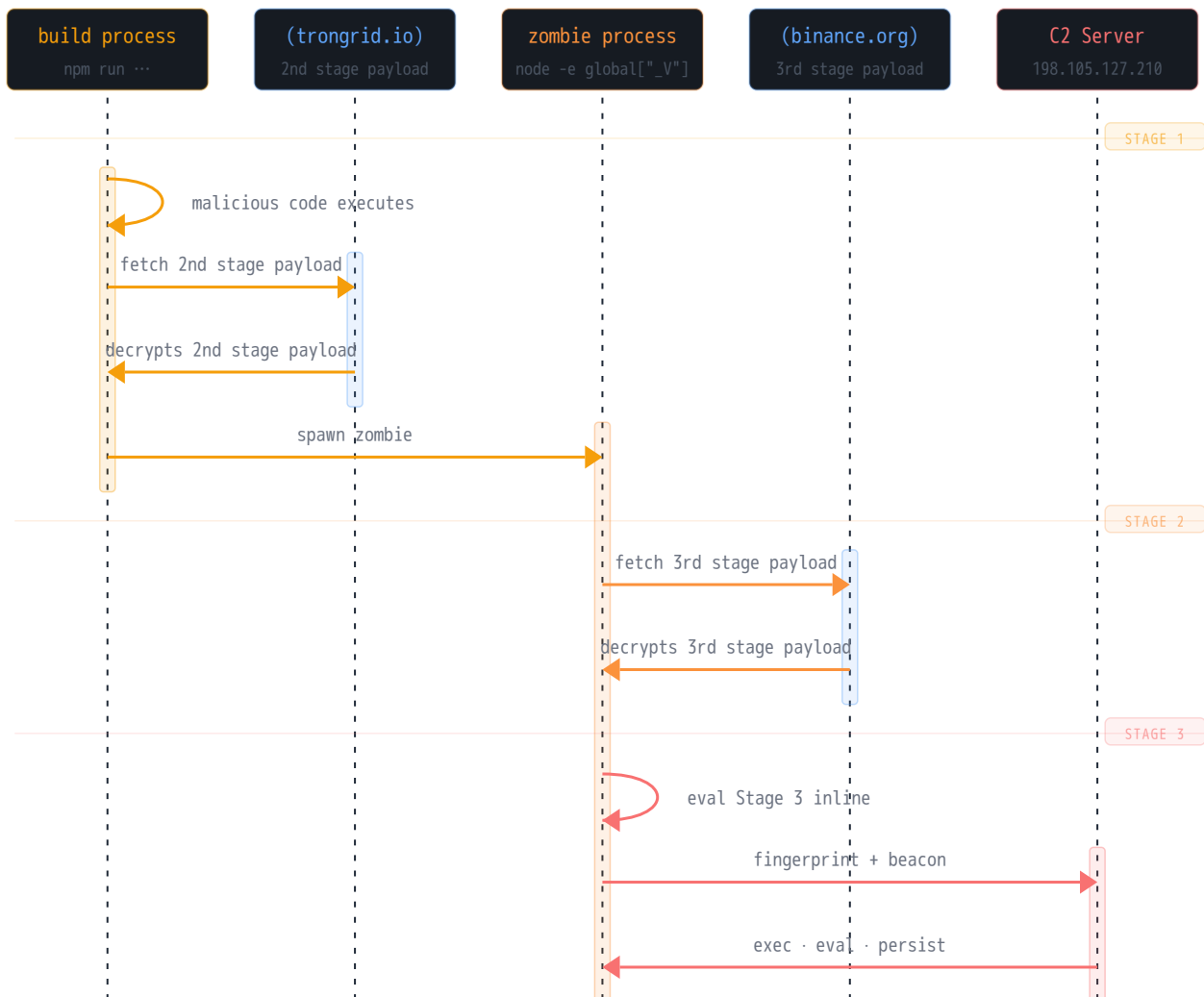
This malicious code is inside the build configuration files, such as `next.config.mjs` or `vue.config.js`. This provides the attacker two benefits:

1. It's harder to spot for a human once it's merged because typically developers rarely change root-level build configurations.
2. If the externally-provided PR is ever permitted to run in CI/CD, the CI/CD environment becomes infected.

Let's analyze what this weirdly obfuscated code does. To the untrained eye or a tired maintainer scrolling through 50 changed files, this might look like a random, minified build artifact. (*I truncated the code to avoid accidental copy/pasting but you can find the full code in the linked PRs above.*)

```
global['_V']='8-1447';global['r']=require;if(typeof module==='object')global['m']=module;(function
```

This malicious snippet siphons out your environment variables and establishes a connection to a "command and control server", which allows the attacker to send arbitrary commands to execute locally.



This malicious snippet runs the moment somebody starts their CI process (`npm run build`) or their dev server (`npm run dev`). Then, it proceeds to download a 2nd stage payload, which is another set of malicious code. This

2nd stage malicious code is executed in a zombie process. This zombie process then downloads a 3rd stage payload, which includes a 91KB javascript code snippet that spins up the command and control server connection which ultimately compromises the machine.

[Stage 1: Step-by-step explainer of what the malicious code actually does](#)

Let's first prettify the code and then analyze it section by section.

Step 1 — Global hooks

Planting flags before anything else runs

The first three lines execute before the main IIFE. They stamp a version marker onto the global object (`_V = '8-1447'`, a campaign ID the C2 server uses to track which payload version infected each machine), then capture require and module into globals so they stay accessible inside the sandboxed function scope that follows. `GhP = 764 - 753` is a simple trick: compute 11 at runtime rather than writing the literal, so `grep` for '11' finds nothing.

Lines 1–7

Step 2 — The shuffler

MDy: a deterministic character scrambler

MDy is the obfuscation engine. Given any string, it performs a deterministic swap sequence across all character positions driven by the hardcoded seed 1111436. The same seed always produces the same permutation, so it is fully reversible at runtime. To a code reviewer it looks like meaningless arithmetic on a character array. Every sensitive string in this file passes through MDy before it is used.

Lines 9–25

Step 3 — The signature string

lcdmccutnorbjrothxgunkypaivtswrsozqf decrypts to 'constructor'

tgr is the 37-character string that has become this malware family's fingerprint across 30+ infected repositories. When passed through MDy and trimmed to GhP characters (which equals 11 at runtime), it produces the string 'constructor'. That word is how the attacker reaches into the JavaScript engine itself without ever writing it in plain text.

Lines 27–28

Step 4 — Function constructor chain

Building executable code without writing eval or Function

`MDy[tgr]` resolves to `MDy['constructor']`, which is JavaScript's built-in Function constructor. Identical to writing `Function()` directly, but invisible to any static scanner searching for that keyword. `oWZ(kcL, MDy(ruc))` decodes

the 890-character `ruc` blob and passes it as the body of a brand-new function, `yus`. `yus` is the custom decoder that only exists in memory at runtime and leaves no trace on disk.

Lines 30–33

Step 5 — Execution

`tzo(5471)` fires and the machine is compromised

`yus` decodes the 2,634-character `quw` blob into valid JavaScript. AoT (an alias for the Function constructor established on line 32) wraps it into a callable named `tzo`. `tzo(5471)` runs it. From this point the machine is compromised: the payload queries a blockchain dead drop for an encrypted Stage 2 loader, spawns a detached child process that survives after the build exits, and the infection continues silently.

Lines 35–40

The `tzo(5471)` call fires a decoded function body that only existed in memory. Here is that logic, annotated.

Step 1 — String array bootstrap

Every API name is hidden behind an index

`_$af402005` is the same character-shuffle algorithm from Stage 1, seeded with 1111436. It is called immediately with a 890-char encoded string and returns `_$_4eb3`: an array of ~55 decoded strings holding every API name, method, and constant used in this function. Writing `_$_4eb3[0]` instead of a literal string means every grep for API names returns nothing. The entire call surface is invisible to static analysis.

Lines 2–29

Step 2 — HTTPS fetcher

HTTP client used to reach the blockchain dead drops

Function `o` is a Promise wrapper around Node's `https` module, accessed as `d(_$ _4eb3[9])` where the string 'https' is decoded at runtime. It makes a GET request, accumulates the response body, and returns parsed JSON. This is the function that reaches `api.trongrid.io` and `fullnode.mainnet.aptoslabs.com`. Both are legitimate, publicly trusted blockchain APIs, which means the outbound request looks completely normal to any network monitor.

Lines 39–49

Step 3 — JSON-RPC caller

JSON-RPC POST client with hostname encoded at runtime

Function `c` makes JSON-RPC POST requests to a blockchain node. The method name is `_$_4eb3[33]` and the target hostnames are `_$_4eb3[34]` and `_$_4eb3[35]`, all decoded at runtime from the string array. Neither the method nor the endpoints are readable without decoding first. This function is called after the TronGrid or Aptos lookup has returned a transaction reference, using that reference to retrieve the encrypted payload from the node.

Lines 71–97

Step 4 — Blockchain dead drop

trongrid.io primary, aptoslabs.com fallback, then XOR decrypt

Function `t` is the payload retrieval engine. The primary path queries `api.trongrid.io` with the address in argument `t`. If that throws, the fallback queries `fullnode.mainnet.aptoslabs.com` with address `e`. The result from either path is used as input to function `c`, a JSON-RPC call to an encoded blockchain node hostname. The raw bytes returned are then XOR-decrypted using key `a`, character by character, producing live executable JavaScript in memory.

Lines 99–114

Step 5 — Zombie spawn

Spawning a zombie process that survives after the build exits

`d(_$4eb3[52])[_$4eb3[51]]()` resolves to `require('child_process').spawn()`. Three flags make the child a zombie: `detached: true` promotes it to its own process group so it survives when the parent exits; `$_4eb3[50]` decodes to `'ignore'`, severing all stdio pipes so the parent does not wait; `windowsHide: true` suppresses any console window on Windows. The build exits with code 0. The developer sees success. The spawned node process runs the Stage 2 payload independently and silently.

Lines 161–166

Finally `tzo(5471)` spawns a zombie process that lives on even after the main process (e.g. `npm run build / npm run dev`) from stage 1 exits.

Running a `ps aux | grep "node -e.*global\['_V'\]"` will show you the zombie process. (*Again, I truncated the code for safety but you can find the full code in the linked PRs above.*)

```
node -e global['_V']='8-1447';if('function' ===typeof require)global['r']=require;if(typeof mod
```

[Stage 2: Loading the command and control payload](#)

Stage 2's code is almost **logically identical** to Stage 1's code. The primary difference is that Stage 2's code is obfuscated with a different seed and loads the 3rd stage payload from a different blockchain address.

Stage 1 (outer wrapper) ← same algorithm → Stage 2 (zombie wrapper)

Shuffler function nameSeed constantSize obfuscation (both = 11)Modulus constantsSignature stringConstructor
accessorEncoded payload blob

– Stage 1 (outer wrapper)+ Stage 2 (zombie wrapper)

```
1global['_V'] = '8-1447';
```

```

-2global['r'] = require;

+2if ('function' === typeof require) global['r'] = require;

3if (typeof module === 'object') global['m'] = module;

+4var a0n, a0b, _global, a0a;

4(function() {

-5 var VRG = "",

+6 var rms = "",

-6 GhP = 764 - 753;

+7 IaL = 200 - 189;

-7 function MDy(f) {

+8 function gOe(z) {

-8 var r = 1111436;

+9 var y = 3547675;

-9 var w = f.length;

+10 var w = z.length;

-10 var h = [];

+11 var m = [];

-11 for (var q = 0; q < w; q++) {

+12 for (var x = 0; x < w; x++) {

```

This 3rd payload is significantly larger at 91 KB and houses the command and control logic. This payload is fetched from Binance Smart Chain (BSC).

```

POST https://bsc-dataseed.binance.org/
Content-Type: application/json

{
  "jsonrpc": "2.0",
  "method": "eth_getTransactionByHash",
  "params": [". . . . ."],
  "id": 1
}

```

Stage 3: The command and control payload

Stage 3 is 91 KB of RC4-encrypted JavaScript, packed with 1,467 encrypted strings and a flattened state machine. Once unpacked, it does three things in sequence:

System fingerprinting. Before contacting the attacker, it takes a snapshot of the compromised machine: OS type and version, CPU count, total RAM, hostname, username, home directory, and all environment variables visible to the Node process. This snapshot is used to uniquely identify each victim.

Establishing a connection to the command and control server `198.105.127.210/socket.io/`. The malware contacts a command and control server via Socket.IO's HTTP long-polling transport. It sends an identification beacon (the system fingerprint) and enters a polling loop, waiting for instructions. The use of Socket.IO over port 80 makes this traffic indistinguishable from normal web application traffic.

Executing arbitrary commands. The C2 server can now issue four classes of commands to any connected victim: run shell commands (`exec`), evaluate arbitrary JavaScript in-process (`eval`), download and execute additional payloads (`download`), and install persistence mechanisms (`persist`). From this point, the attacker has full interactive access to anything the Node process can reach.

The immediate targets are environment variables: `AWS_ACCESS_KEY_ID` , `DATABASE_URL` , `STRIPE_SECRET_KEY` , `OPENAI_API_KEY` , `GITHUB_TOKEN` , and anything else present in `.env` files at build time. SSH keys, browser credential stores, and shell history are secondary targets, accessible on command.

Machine compromised.

Once the malicious payload executes and compromises a machine, the true danger of a supply chain attacks are second-order effects.

In a CI/CD environment, the blast radius is even larger. Many pipelines run with elevated IAM roles that have broad access to cloud infrastructure. A single infected build step (even in a dev environment) in a pipeline can expose database credentials, artifact signing keys, AWS roles with `AdministratorAccess` , and internal service tokens.

Another risk of lateral movement is the spread to other packages that depend on a compromised package. A historical example: In 2018, the popular npm package `event-stream` (which had over 1.5 million weekly downloads at the time) was compromised. The original author, lacking the time to maintain it, [handed ownership over](#) to a seemingly helpful contributor named "right9ctrl".

The attacker added a new dependency called `flatmap-stream` , which they also controlled. Then, they pushed an update to `flatmap-stream` containing a highly targeted payload designed to steal cryptocurrency from the bitcoindash app.

Because `event-stream` was a foundational dependency used by thousands of other packages (like Vue CLI), millions of developers downloaded the compromised code. The attack went undetected for nearly two and a half months.

In an effort to illustrate the impact of a supply chain attack, I created the following impact calculator using Gemini Deep Research.

Interactive Tool

Supply Chain Impact Calculator

Adjust the parameters to model a real attack scenario. Heuristics provided by Gemini Deep Research.

Primary Machines Compromised

640

Developer machines & CI runners with malicious code executed

Second-Order Victims

32,000

End-user systems reached via 32 poisoned publishing pipelines

[Blockchains as a permanent, public hosting infrastructure for malicious code](#)

Traditional malware relies on infrastructure that can be dismantled. Security teams and law enforcement can identify a malicious C2 server, file an abuse report, and have it taken offline within hours. Domain registrars can revoke DNS. ISPs can null-route IP addresses. Cloud providers can suspend accounts. This response chain, while imperfect, has historically been an effective backstop against persistent malware campaigns.

Blockchain breaks this model entirely.

When a malware author stores an encrypted payload inside a blockchain transaction, that data becomes permanent and globally replicated. A blockchain network runs across thousands of independent nodes worldwide. No single authority (not any exchange, not law enforcement, not any government) can alter or remove a confirmed transaction. The data is as permanent as the blockchain itself, which is designed specifically to be immutable.

In contrast, the `axios` attack from a few days ago could be taken down because the 2nd stage payload was hosted on a GitHub repository. This wouldn't be possible with the attempted attack on `better-auth`.

Looks like the postInstall setup script grabbed a second stage from GitHub repos which have been taken down so the threat should be neutralized as it stands now.

We are working it, sharing what we know as of now - gist.github.com/joe-desimone/3...

[About Casco](#)

Casco performs always-on autonomous security testing on your applications, APIs, infrastructure, and AI systems. Talk to us if you want to get hacked before bad actors hack you. [Book a demo](#).

This blog post wouldn't be possible without the help of

- [Bereket Engida](#) and the [Better-Auth](#) team
- [Blaxel](#) for providing a secure sandbox for us to analyze the malicious code
- [Noah Lebovic](#) for providing valuable feedback on the blog post

Source: <https://casco.com/blog/the-blueprint-of-a-north-korean-attack-on-open-source>