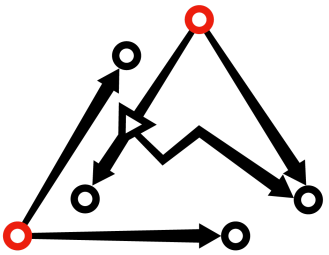

Least Authority

PRIVACY MATTERS

tBTC Bridge V2

Security Audit Report

Keep Network

Updated Final Audit Report: 29 September 2022

Table of Contents

Overview

Background

Keep Network has requested that Least Authority perform a security audit of their tBTC Bridge v2.

Project Dates

- **May 02 - July 20:** Initial Code Review *(Completed)*
- **July 22:** Delivery of Initial Audit Report *(Completed)*
- **September 19:** Verification Review*(Completed)*
- **September 20:** Delivery of Final Audit Report *(Completed)*
- **September 29**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Alicia Blackett, Security Researcher and Engineer
- Alejandro Flores, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- Giorgi Jvaridze, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the tBTC Bridge v2 followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Sortition Pools:
  https://github.com/keep-network/sortition-pools
- Random Beacon:
  https://github.com/keep-network/keep-core/tree/main/solidity/random-beacon
- ECDSA:
  https://github.com/keep-network/keep-core/tree/main/solidity/ecdsa
- Solidity:
  https://github.com/keep-network/tbtc-v2/tree/main/solidity
- Bitcoin SPV
  https://github.com/Keep-network/bitcoin-spv

Specifically, we examined the Git revisions for our initial review:

Sortition Pools: `b3fcb43e09d9fc2304fa2f65635aec406a6cfe79`

Random Beacon: `c2c77c561745a5c3c46eed8e3900a5c83c798265`

ECDSA: `c2c77c561745a5c3c46eed8e3900a5c83c798265`

Solidity: `a038deeb55891909c36a8537ad390e1e5da3fd24`

Bitcoin SPV: `c4788e9fc07bc193872a462e186fb02c9d215372`

For the verification, we examined the Git revisions:

Sortition Pools: `08e6252a38d4f16ccf56acc67517bf79468e631d`

Random Beacon: `dad00020f0365617319636e629e2c113bfb720d8`

ECDSA: `dad00020f0365617319636e629e2c113bfb720d8`

Solidity: `c10b380c2506047cf01e094df9b402c55c120584`

Bitcoin SPV: `856849612ef49114af18c0f407eaa74afc2ee4be`

For the review, these repositories were cloned for use during the audit and for reference in this report:

Sortition Pools:
https://github.com/LeastAuthority/Keep-Network-Sortition-Pools

Random Beacon and ECDSA:
https://github.com/LeastAuthority/Keep-Network-Keep-Core

Solidity:
https://github.com/LeastAuthority/Keep-Network-tBtc-v2

Bitcoin SPV:
https://github.com/LeastAuthority/Keep-Network-Bitcoin-Spv

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- `May-July_2022__Final_v2_audit_Least_Authority.pdf` (*shared with Least Authority via Slack on 2 May 2022*)

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service attacks and security exploits that would impact or disrupt the execution of the smart contracts;
- Vulnerabilities in the smart contracts' code;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of the Keep Network's tBTC Bridge v2, including its system components: Bitcoin SPV, sortition pools, random beacon, and ECDSA. The tBTC Bridge is a decentralized autonomous organization (DAO) that manages and oversees a cross-chain atomic swap setup to allow the use of Bitcoin to secure Ethereum assets. Our team examined the tBTC Bridge v2 implementation for errors, security vulnerabilities, and attack vectors in both the v1 and v2 token implementations and the network's typical operation.

The tBTC Bridge v2 is designed as a trustless and decentralized bridge between Bitcoin and Ethereum to facilitate the use of Bitcoin on the Ethereum blockchain without the need to trust a third party. This is achieved through a process of users depositing and revealing BTC, for which coins are subsequently minted on the Ethereum Network to match the BTC revealed by the depositor. This process has many moving parts and must be carefully implemented. Our team considered several different broad-attack categories during the course of our audit, including known security vulnerabilities in Bitcoin and Ethereum.

System Design

The tBTC Bridge v2 is composed of subsystems, with the Bridge smart contracts being the hub to all of the component operations. Our team examined the design of each of these components to identify potential security vulnerabilities that would affect the system or its users. We also investigated assumptions that the system's security is built upon as well as features that could increase the attack surface of the system.

The Bridge

The Bridge smart contract is the core component of the tBTC Bridge v2 and, as such, has considerable functionality wrapped into it. The Bridge imports lower level libraries with the `using` keyword. This design approach allows the Bridge to declare smaller functionality in a succinct manner that can be organized into lower units of abstraction.

The Bridge smart contracts declare a wide set of operating parameters for the network's associated fees and rates. These values are set in the contracts and can only be changed by the governance DAO. As a result, the DAO is the sole caretaker of the Keep Network, a governance strategy consistent throughout the implementation.

The Bridge handles deposit and redemption flows at a high level and binds those services to governance giving it more precise control of the network. It also defines the process for fraud claims in the network and the structure of a valid heartbeat message. We analyzed these processes for a series of vulnerabilities, including fraud proof and inactivity claim abuse, Denial of Service attacks, reentrancy issues, permissions, and protocol correctness. We identified one issue relating to the deposit flow and Bitcoin SPV transactions (Issue B).

Maintainers

The `MaintainerProxy` smart contract defines functions that off-chain clients call to interact with the network. Maintainers must be approved by the DAO before they can call these functions. Maintainers are charged with carrying out network maintenance jobs that are separate from the wallet maintenance tasks. As a result, they are reimbursed for transactions they carry out on behalf of the network. We reviewed the `MaintainerProxy` smart contracts for a series of vulnerabilities, including reentrancy, permissions, and correct implementation and did not identify any vulnerabilities.

The Bank

The Bank component is responsible for tracking Bitcoin balances in the tBTC Bridge v2. The Bridge is the only component that is allowed to increase balances in the Bank, a constraint enforced in the Bank smart contracts by the modifier `onlyBridge`. The Bank controls all minting and unminting operations through the tBTC token smart contract. We examined the Bank for a series of vulnerabilities, including reentrancy, permissions, and correct implementation, and identified an issue relating to updates between non-zero account balances (Issue A).

The Vault

The Vault is an interface fulfilled by the `DonationVault` and the `TBTCVault` smart contracts. The `TBTCVault` smart contract is responsible for minting and unminting tBTC in a 1:1 relationship with Bitcoin. The Bank is the only contract that is allowed to call these vault contracts, a constraint enforced by the modifier `onlyBank`. We examined the vaults for a series of attack vectors, including reentrancy, permissions, and correct implementation, and did not identify any vulnerabilities.

Sortition Pool

The sortition pool tracks nodes and their respective stakes and weights them according to that stake in the system. The sortition system is one of three critical elements underpinning the tBTC Bridge v2's major security assumptions and handles the staking and unstaking process. Undue influence in the sortition pool could facilitate attacks across several areas of the network, including bridge operations, rewards eligibility, and wallet maintenance. We examined the sortition pool for a variety of attack vectors, including possible ways to fraudulently withdraw funds, reentrancy errors, and correctness of the underlying algorithm and did not identify any vulnerabilities.

Market Assumptions

Several aspects of the tBTC Bridge v2 are sensitive to external market forces, such as Ethereum's price and gas cost, Bitcoin's price and transaction fees, and the frequency of necessary internal maintenance actions. Key parts of the network rely on these market forces being within a normal range. We analyzed these vectors for attacks and manipulation but could not predict security issues that may arise at boundary conditions in one or more of these underlying markets. For example, when wallets must redistribute their funds, the number of transfers necessary is dictated by the maximum amount allowed per payment. If this parameter is increased, uneven distributions of funds in a given group could eventually occur. If it is set too low, the gas costs incurred by the number of transactions will start to become prohibitively expensive. The price point where that inflection point sits is sensitive to the cost of gas in the Ethereum network. In the worst case scenario, this could result in the DAO being drained of funds faster than it is able to replenish them over a given period of time. Although there are adjustable parameters accessible to the governance DAO that could mitigate these forces, as with all markets, excessive reliance on governance can have unexpected secondary consequences or surprising trade-offs.

The Heartbeats Mechanism

Heartbeats in the tBTC Bridge v2 act as a reputation system. They determine who is likely to respond, when necessary, to sign transactions. For every wallet, the tBTC Bridge v2 forms a 51/100 threshold signing scheme. Heartbeat checks are intended to ensure that the network can always sign transactions.

Heartbeats are signed messages sent at a regular interval across the network to the rest of the nodes signing group, proving that the keys are still available and online. If a node misses a heartbeat, it can be penalized by the other group members. To start this penalty process, an operator must file an inactivity claim against the accused node. If the challenge is not defeated within a set time frame, the node gets punished. If the challenge is defeated by receiving another heartbeat, the node remains in good standing.

These penalties can range from slashing its rewards for participating in the network for a period of time all the way to dropping the node from the group entirely.

We examined the heartbeat system for a series of attacks, including Denial of Service vectors, misaligned incentives, fraudulent claims, and inactivity claim abuse and did not identify any vulnerabilities. However, as with all reputation systems, we urge further modeling and close monitoring of this mechanism in production to check for suspicious activity including, but not limited to, relatively high request volumes from a node and failed malformed requests.

Wallet Redistribution

The network has a configurable parameter that determines how many nodes in a given group of 100 must be online to keep the wallet active. If the number of nodes falls below this number, then the network triggers a redistribution of that wallet's funds. We examined this critical functionality closely for attack vectors and looked for ways in which actors could abuse the inactivity claim process, denial of service attacks, protocol correctness, and attacks related to outsized wallet influence. We found no direct vulnerabilities but recommend taking caution and further monitoring this functionality and the rest of the heartbeat system to verify that gas costs and the resulting distribution sums remain within reasonable bounds.

Staking and Slashing

We examined attack vectors related to the staking and slashing processes as well as the delegation and un-delegation of funds.

The inactivity claims and fraud proof system is a key component of the security assumptions of the staking and slashing process. We closely examined this system for incentive alignment, protocol correctness, and possible abuse or manipulation tactics and found none specific to the tBTC Bridge v2. However, there are theoretical attacks in this area that exist on Ethereum, and are known as fraud-proof attacks, which fall under a broad type of censorship attack in proof-of-stake systems. As a mitigation to these types of attacks, the Keep Network team has chosen reasonable values for the number of block confirmations that wallets wait to receive before acting on the Bitcoin and Ethereum side.

Rewards are paid out to operators in a wallet by the sortition pool. Given that operators are awarded tokens proportional to their stake, Sybil attacks are prevented as a result of identity being limited and costly in this system. However, this introduces a size and number issue for wallets. If a node has outsized influence in a group, it could exert its influence over that group at signing time by withholding signatures to prevent a majority or otherwise affect the outcome of the signature process. The Keep Network team has modeled this problem space and, while we consider their assumptions sound, and the realization of the issue highly unlikely, the possibility of large or numerous wallets colluding to affect the network adversely remains.

Democratic Enforcement

Groups are free to select a process to check heartbeats within the group, which makes the network open to group collaboration. This has both benefits and disadvantages, as group members could arbitrarily punish other group members, but they could also adapt and recognize bad behavior that the network at large is too slow or unwilling to identify. Furthermore, group members have a strong incentive to work together because they are rewarded.

Bitcoin SPV

In the tBTC Bridge v2, Bitcoin must be deposited into the newest Keep wallet and then revealed to the rest of the network by crafting a reveal transaction, which is then broadcast in an event showing that a deposit has been revealed. The Keep Network team uses a Bitcoin SPV scheme to verify the payment in a timely

fashion. Bitcoin SPV is a well-understood payment scheme that is mentioned in the original Bitcoin whitepaper.

The SPV process is battle-tested, but it does introduce specific security concerns to users. During the course of our audit, we discovered a vulnerability in Bitcoin SPV proofs, which affects the `bitcoin-spv` library that the tBTC Bridge v2 uses. A valid SPV proof can be constructed for a fake transaction. This would defeat the underlying security assumptions that the deposit process is built upon and potentially allow tBTC tokens that have no real backing to be minted. We recommend that a check be performed to identify and discard invalid transactions for every 64-bit node, or that the complexity of the address space be increased to make brute-forcing of the SPV proof hash infeasible (Issue B).

Although the Bitcoin SPV repository was out of scope for this review, the bitcoin data parsing functions in the Bitcoin SPV repository, that are used by `tBTC-v2/solidity`, were in scope. Our team looked specifically at functions from the `BTCUtils`, `Bytes`, `ValidateSPV` and `CheckBitcoinSigs` libraries, which are critical in correctly interpreting the Bitcoin raw transaction data. Our team noted that the Keep Network team used the correct endianness when choosing functions that interpret data from these transactions and made correct reversals when necessary. We did not identify specific vulnerabilities within these functions.

Governance

The governance component pays for certain maintenance transactions necessary to continue running the network. For example, the wallet redistribution, donation, and redemption flows incur costs to the governance component. Although we neither identified a specific attack vector utilizing governance functions nor any vulnerabilities that could result in the draining of governance funds, an attack targeting governance functionality is always possible due to the system's design and should be carefully monitored. Any area where governance funds are used or where incurred fees have a high incentive to be avoided should be studied carefully.

Code Quality

The repositories in scope are implemented in accordance with accepted engineering standards and generally follow best practices as demonstrated by modules following a logical flow, the correct use of primitives, and the efficient utilization of modifiers. The smart contracts in the codebase are ordered systematically and show a high degree of organization. The Bridge components follow a clean pattern for importing libraries.

Our team identified some areas of improvement that would increase the overall quality and security of the code. The variable defined to store the Unix timestamp will overflow at a known point in the future. When this occurs, a system could be disabled, which could result in the assets held by the network becoming unretrievable. We recommend that an alternative variable type be used to avoid this (Suggestion 5). In addition, the Bank and tBTC smart contracts do not adhere to the EIP-2612 standard. This could result in a possible incompatibility with third-party libraries interacting with these contracts. We recommend modifying the smart contracts to be compliant (Suggestion 6).

We identified variables that are incorrectly defined where best practice guidelines recommend that the variables be defined as immutable. We recommend that these variables be defined appropriately (Suggestion 1). In addition, we identified instances where the definition of variables can be improved to reduce gas costs (Suggestion 2). We also recommend that custom errors be used to reduce gas consumption (Suggestion 3).

The implementation defines a variable for identifying the caller `msg.sender`, which is not necessary and increases gas consumption. We recommend using the caller address directly (Suggestion 4). Finally, we

identified a redundant assertion that increases gas consumption. We recommend that the assertion be removed (Suggestion 7).

Tests

The tBTC Bridge v2 implements a comprehensive test suite and makes sufficient use of linting, formatters, and continuous integration and delivery pipelines. The project uses Hardhat to provision and run a solidity test environment. We were able to install necessary dependencies and run the full test suite and linters for the project without difficulty.

The test suite includes unit tests, integration tests, and end-to-end flows testing all key processes, including deposits, redemptions, wallet creation, and fund transfers. All major, and some minor, components have their own test suites in a similarly robust approach, with mocking and unit tests prevalent throughout their tests. Our team suggests some minor improvements to the test suite (Suggestion 8).

The test framework created by the Keep Network team is robust, uses mocked components of the system appropriately, and includes simulations of wallet closures and fund redistribution. The file `simulation/wallets.js` demonstrates that the wallet lifecycle has been modeled to monitor and analyze specific costs associated with redistribution. We commend this modeling and analysis and encourage further modeling and analysis in the future.

Documentation

Our team was provided comprehensive project documentation, which sufficiently describes all major components of the system. Our team found this documentation to be accurate and helpful in understanding, and reasoning about the security of, the system. The Keep Network team has a system in place for tracking RFCs, which explain design decisions that were included in the repository in scope. The RFCs allowed our team to analyze the design and implementation of the network at a deeper level.

Code Comments

The tBTC Bridge v2 codebase is generally well commented. The code comments are up to date, succinct, and focus on the purpose instead of only the description. The solidity contracts have sufficient code comments that typically signal an entry point into the system, allowing auditors to quickly understand the interface of each smart contract.

Scope

The scope for this security review was comprehensive and included all security-critical components.

Dependencies

We analyzed project dependencies for vulnerabilities, out-of-date packages, and excessive or unnecessary dependency usage. We found no issues, within dependencies, that presented security issues. Additionally, we ran the standard suite of static analysis tools against the dependencies used and found no issues.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS

Issue A: Updates between Non-zero Allowances Can Result in Exploits	Resolved
Issue B: Bitcoin SPV Merkle Proofs Can Be Faked	Resolved
Suggestion 1: Use Immutable Instead of Storage Variable	Resolved
Suggestion 2: Use Fit Data Types for Storage Variables	Will Not Fix
Suggestion 3: Use Custom Errors to Save Gas	Will Not Fix
Suggestion 4: Use Caller Directly	Partially Resolved
Suggestion 5: Expand Data Type for Timestamp	Will Not Fix
Suggestion 6: Make Bank.sol and TBTC.sol Compliant with EIP-2612	Resolved
Suggestion 7: Remove Redundant Code	Resolved
Suggestion 8: Improve Test Suite	Unresolved

Issue A: Updates between Non-zero Allowances Can Result in Exploits

Location

`solidity/contracts/bank/Bank.sol#L122-L124`

Synopsis

The process by which a user updates allowances from non-zero to non-zero can be susceptible to exploits.

When user A approves the transfer of N tokens to user B, and user A updates the allowance to M using the `approve` function, user B can deploy the `approve` transaction in the mempool and take M+N tokens by transferring N tokens just before the second approval with a higher gas price, and transferring M tokens after the second approval.

Impact

In this case, this issue could result in User B stealing User A's M tokens.

Remediation

We recommend that the Keep Network team prevent updates between non-zero allowances. The user planning to update the approval from non-zero to non-zero must set the allowance to zero first. As a result, the user can detect if the allowance was used by the approved user before the new approval. For example:

```
function approveBalance(address spender, uint256 amount) external {
      require(!((amount!= 0) && (allowance[msg.sender][_spender] != 0)));
      _approveBalance(msg.sender, spender, amount);
  }
```

Status

The Keep Network team has added the `require` function as suggested to prevent the exploit.

Verification

Resolved.

Issue B: Bitcoin SPV Merkle Proofs Can Be Faked

Location

[Keep-Network-Bitcoin-Spv]

Synopsis

[Issue #192 in the summa-tx/bitcoin-spv repository] provides a good summary of the Issue. An attacker can create a valid SPV proof out of a fake transaction. The attacker could use this SPV proof during the deposit flow to trick the Bridge into minting tokens that they could then withdraw.

Impact

If an attacker is able to fake an SPV transaction deposit, then the Bridge could be tricked into minting an unbacked balance of new tokens.

Preconditions

The attacker must successfully brute-force a hash and then use it in an SPV proof. Additionally, the tBTC Bridge v2 must be using a version of `bitcoin-spv` that does not check for the malformed Merkle proof that this attack uses.

Feasibility

This attack would be quite expensive upfront, requiring an estimated several million USD to carry out — depending on current market prices. Additionally, the hardware requirements would be substantial, likely requiring state-of-the-art miners capable of brute-forcing the necessary hashes in a reasonable time frame. However, depending on the price of Bitcoin, and as the Keep Network and its average wallet sizes grow, there will be a point where this attack will become profitable after the initial costs.

Technical Details

Simple payment verification relies on Merkle proofs created with the previous transaction's hash included to verify provenance. The problem is that bitcoin does not differentiate between inner nodes and leaf nodes in these proofs. As a result, a malicious actor can create a transaction with a brute-forced inner node in a Merkle proof that matches the expected values of another arbitrarily selected valid transaction. This requires brute-forcing a total of 72-bits of address space in order to compute the necessary bytes to make the constructed proof match. With this Merkle tree available, the attacker can submit a transaction with an arbitrary amount of Bitcoin to a victim SPV wallet. The transaction would be seen as a valid proof and the tBTC Bridge v2 would trigger a deposit, eventually minting tokens at an equal proportion to the amount in the malicious transaction. The attacker would then trigger a redemption process as soon as possible and retrieve a balance that they never actually owned.

Remediation

We can recommend two mitigations for this issue: A quick solution can be achieved by checking that every 64-bit node in the Merkle proof is not a valid transaction and either discarding it or further validating it with the knowledge that it is suspicious. Another more expensive approach is to send a Merkle proof of

the coinbase transaction, along with the SPV proofs for the transaction, which would add sufficient complexity to the address space necessary to brute-force to make this attack infeasible.

Status

The Keep Network team has implemented the validation and added tests to mitigate and prevent this attack.

Verification

Resolved.

Suggestions

Suggestion 1: Use Immutable Instead of Storage Variable

Location

contracts/EcdsaDkgValidator.sol#L67

contracts/WalletRegistryGovernance.sol#L86

Synopsis

The `sortitionPool` and `walletRegistry` variables are not updated but are set on deployment only once.

Mitigation

We suggest using the `immutable` definition, instead of the `storage` definition, to save gas.

Status

The Keep Network team has started using the `immutable` variable instead of `storage`.

Verification

Resolved.

Suggestion 2: Use Fit Data Types for Storage Variables

Location

ecdsa/contracts/WalletRegistry.sol#L72

ecdsa/contracts/WalletRegistry.sol#L77

ecdsa/contracts/WalletRegistry.sol#L84

ecdsa/contracts/WalletRegistry.sol#L89

ecdsa/contracts/WalletRegistry.sol#L94

ecdsa/contracts/WalletRegistry.sol#L99

ecdsa/contracts/WalletRegistry.sol#L107

ecdsa/contracts/WalletRegistryGovernance.sol#L28-L29

ecdsa/contracts/WalletRegistryGovernance.sol#L32

ecdsa/contracts/WalletRegistryGovernance.sol#L35

ecdsa/contracts/WalletRegistryGovernance.sol#L38

ecdsa/contracts/WalletRegistryGovernance.sol#L41

ecdsa/contracts/WalletRegistryGovernance.sol#L44

ecdsa/contracts/WalletRegistryGovernance.sol#L47-L81

ecdsa/contracts/WalletRegistryGovernance.sol#L84

ecdsa/contracts/WalletRegistryGovernance.sol#L88

Synopsis

These variables could fit into smaller data types and be packed together with other variables for gas optimization.

Mitigation

We suggest using fit data types for storage variables.

Status

The Keep Network team has acknowledged the issue but decided not to address this suggestion, as the suggested mitigation could cause the smart contract size to exceed the maximum contract size limit.

Verification

Unresolved.

Suggestion 3: Use Custom Errors to Save Gas

Location

Examples (non-exhaustive):

contracts/bridge/Bridge.sol#L250-L253

contracts/bridge/Bridge.sol#L256

Synopsis

The codebase currently uses `require` and `revert` string messages for error handling. However, using a `revert` with an Error event, instead of a string error message in transactions, considerably optimizes gas costs, according to Solidity documentation recommendations.

Mitigation

We recommend using a `revert` with an Error event to handle errors.

Status

The Keep Network team has acknowledged the issue but decided not to address this suggestion, as the framework they are using does not support custom errors.

Verification

Unresolved.

Suggestion 4: Use Caller Directly

Location

contracts/libraries/EcdsaAuthorization.sol#L178

contracts/libraries/BeaconAuthorization.sol#L175

contracts/libraries/BeaconAuthorization.sol#L473

contracts/vault/DonationVault.sol#L59

contracts/vault/TBTCVault.sol#L94

Synopsis

These memory variables are not needed. The caller reference is more cost-effective than `load`.

Mitigation

We suggest using the caller directly.

Status

The Keep Network team has stated that in-memory variables in `EcdsaAuthorization` and `BeaconAuthorization` are used deliberately to clearly define the provider and operator roles. For `TBTCVault` and `DonationVault`, a caller variable is now used.

Verification

Partially Resolved.

Suggestion 5: Expand Data Type for Timestamp

Location

contracts/Rewards.sol#L43

contracts/bridge/Deposit.sol#L88

contracts/bridge/Deposit.sol#L97

contracts/bridge/Fraud.sol#L68

contracts/bridge/MovingFunds.sol#L87

contracts/bridge/Redemption.sol#L148

contracts/bridge/Wallets.sol#L76

contracts/bridge/Wallets.sol#L79

contracts/bridge/Wallets.sol#L82

Synopsis

As of the 7th of February, 2106, the epoch timestamp will exceed the `uint32` max limit, resulting in an overflow of `uint32`.

Mitigation

We suggest using a larger data type to store the Unix timestamp.

Status

The Keep Network team has stated that the timestamp is represented as `uint32` to achieve gas efficiency and decided not to address this suggestion at the time of this verification.

Verification

Unresolved.

Suggestion 6: Make Bank.sol and TBTC.sol Compliant with EIP-2612

Location

`contracts/bank/Bank.sol#L46`

Synopsis

EIP-2612 defines a standard that enables users to issue token allowances using signatures.

Signed data includes a nonce value that the client can get from the contract by querying the "`function nonces(address owner) external view returns (uint)`" function. This function is part of the EIP-2612 standard.

The Bank smart contract implements EIP-2612 but uses mapping `nonce` (singular) instead of `nonces`. This might create an issue when third-party libraries (that support EIP-2612) try to interact with the Bank and assume that nonces are tracked with `nonces` instead of `nonce`.

The same applies to TBTC, which implements `@thesis/solidity-contracts/contracts/token/ERC20WithPermit.sol#L30`

Mitigation

We recommend that `nonce` be renamed and mapped to `nonces`.

Status

The Keep Network team has updated the code as suggested.

Verification

Resolved.

Suggestion 7: Remove Redundant Code

Location

`contracts/Rewards.sol#L103`

Synopsis

This particular check is always `true` because `uint` values are always equal to or greater than zero. As a result, the check only spends gas.

Mitigation

We recommend removing the codeline.

Status

The Keep Network team has updated the check to verify that the pool weight receives a non-zero value.

Verification

Resolved.

Suggestion 8: Improve Test Suite

Synopsis

To test the tBTC Bridge smart contracts in an Ubuntu 20.04 WSL, some modifications were required for the tests to run:

1. Forcing some NPM libraries to be installed (normal installation would not work)
2. Executing some `hardhat.config.ts` changes:
    a. Adding the parameter `allowUnlimitedContractSize: true` under the Hardhat network config to avoid the test's execution errors
    b. Disabling `contractSizer` to prevent the tests from breaking while trying to read all the contracts in this module

Remediation

We recommend updating the tests so that tests run without modifications.

Status

The Keep Network team has stated that they could not reproduce the problem and, therefore, captured it in a GitHub issue for further investigation.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, and

zero-knowledge protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.