STRONGHOLD
SECURITY

SatsDao Satoshi Security Audit Report

April 20, 2024

Contents

Executive Summary

Title	Description
Client	SatsDao
Project	Satoshi
Platform	Ethereum
Language	Solidity
Repository	https://github.com/satsDAO/Satoshi/, https://github.com/transmissions11/solmate
Initial commit	bb8d6fe1c6c947c95c041fcd9f075fad840771e9, c892309933b25c03d32b1b0d674df7ae292ba925
Final commit	debfd181688fd708be680f788759ff63dbe6f79d
Timeline	April 12 2023 - April 16 2024

Project Overview

The Satoshi Vault Contract implements a vault for the tBTC token where 1 tBTC equals 1e8 SATS (Satoshis). This contract allows depositing and withdrawing tBTC in a way that represents these transactions in SATS, managing the conversion, and keeping track of the net flow of tBTC within the contract.

Audit Scope

File	Link
Satoshi.sol	Satoshi.sol
ERC4626.sol	ERC4626.sol

Audit Methodology

General Code Assessment

The code is reviewed for clarity, consistency, style, and whether it follows code best practices applicable to the particular programming language used, such as indentation, naming convention, commented code blocks, code duplication, confusing names, irrelevant
or missing comments, etc. This part is aimed at understanding the overall code structure and protocol architecture. Also, it seeks to learn overall system architecture and business logic and how different parts of the code are related to each other.

Code Logic Analysis

The code logic of particular functions is analyzed for correctness and efficiency. The code is checked for what it is intended for, the algorithms are optimal and valid, and the correct data types are used. The external libraries are checked for relevance and correspond to the tasks they solve in the code. This part is needed to understand the data structures used and the purposes for which they are used. At this stage, various public checklists are applied in order to ensure that logical flaws are detected.

Entities and Dependencies Usage Analysis

The usages of various entities defined in the code are analyzed. This includes both: internal usage from other parts of the code as well as possible dependencies and integration usage. This part aims to understand and spot overall system architecture flaws and bugs in integrations with other protocols.

Access Control Analysis

Access control measures are analyzed for those entities that can be accessed from outside. This part focuses on understanding user roles and permissions, as well as which assets should be protected and how.

Use of checklists and auditor tools

Auditors can perform a more thorough check by using multiple public checklists to look at the code from different angles. Static analysis tools (Slither) help identify simple errors and highlight potentially hazardous areas. While using Echidna for fuzz testing will speed up the testing of many invariants, if necessary.

Vulnerabilities

The audit is directed at identifying possible vulnerabilities in the project's code. The result of the audit is a report with a list of detected vulnerabilities ranked by severity level:

Severity	Description
Critical	Vulnerabilities leading to the theft of assets, blocking access to funds, or any other loss of funds.
High	Vulnerabilities that cause the contract to fail and that can only be fixed b y modifying or completely replacing the contract code.
Medium	Vulnerabilities breaking the intended contract logic but without loss of fun ds and need for contract replacement.
Low	Minor bugs that can be taken into account in order to improve the overall qu ality of the code

After the stage of bug fixing by the Customer, the findings can be assigned t he following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect it s security.
Acknowledged	The Customer took into account the finding. However, the recommendations wer e not implemented since they did not affect the project's safety.

Findings Summary

Severity	# of Findings
Critical	0
High	1
Medium	3
Low	1

ID	Severity	Title	Status
H-1	High	Everyone can sweep accidental deposits	Fixed
M-1	Medium	The `deposit` can be blocked by `mint`.	Fixed
M-2	Medium	No checking of input variables	Fixed
M-3	Medium	The `mint` function and inconsistent states	Fixed
L-1	Low	Magic constants	Fixed

Findings

Critical

Not Found

High

H-1 High Everyone can sweep accidental deposits Fixed

Description

Satoshi.sol#L190

Everyone can sweep accidental deposits. Thus, an attacker can take other users' tokens.

Recommendation

We recommend adding the `onlyOwner` modifier.

Client's commentary

The `sweep` function was deleted.

Fixed in debfd181

Medium

M-1 Medium The `deposit` can be blocked by `mint`. Fixed

Description

Satoshi.sol#L77

ERC4626.sol#L61

ERC4626.sol#L66

Satoshi.sol#L154

An attacker can use the `mint` function to front-run a first-user deposit. The `mint` function allows receiving shares without transferring assets for input values less than 1e8.

Since `supply` is not 0 and `totalAssets` will be 0 after `mint`, we receive an exception in the `convertToShares` function.

Recommendation

We recommend updating the offset mechanism for the vault.

Client's commentary

Fixed in debfd181.

That reported problematic scenario has been fixed, when small shares are rounded to SAT (1e8) across the `mint`, `previewMint`, `convertShares` functions in the SATS contract.

M-2 Medium No checking of input variables Fixed

Description

- `receiver`:
  Satoshi.sol#L79
  Satoshi.sol#L91
  Satoshi.sol#L104
  Satoshi.sol#L118

Users can lose their funds if they enter their addresses incorrectly.

Recommendation

We recommend adding null address checks.

Client's commentary

Fixed in debfd181.

Null address checks have been added to the SATS contract.

M-3 Medium The `mint` function and inconsistent states Fixed

Description

Satoshi.sol#L89. For example:

```
--- wallet balances ---
[user1] tBTC: 1.0
[user1] SATS: 0.0
[user2] tBTC: 0.5
[user2] SATS: 0.0
--- contract balances ---
[SATS] tBTC: 0.0
```

If user1 mints with the `100000000` input value and user2 mints with the `1` input value, then we will have the following state:

```
--- wallet balances ---
[user1] tBTC: 0.999999999999999999
[user1] SATS: 0.0000000001
[user2] tBTC: 0.499999999999999999
[user2] SATS: 0.000000000000000001
--- contract balances ---
[SATS] tBTC: 0.000000000000000002
```

If user2 mints with the `1` input value and user1 mints with the `100000000` input value, then we will have the following state:

```
[user1] tBTC: 1.0
[user1] SATS: 0.0000000001
[user2] tBTC: 0.5
[user2] SATS: 0.000000000000000001
--- contract balances ---
[SATS] tBTC: 0.0
```

Recommendation

We recommend changing the implementation of the `mint` function.

Client's commentary

Fixed in debfd181. That reported problematic scenario has been fixed, when small shares are rounded to SAT (1e8) across the `mint`, `previewMint`, `convertShares` functions in the SATS contract.

Low

L-1	Low	Magic constants	Fixed

Description

- `1e8`:
  Satoshi.sol#L133
  Satoshi.sol#L145
  Satoshi.sol#L154
  Satoshi.sol#L165
- `1e18`
  Satoshi.sol#L183

Recommendation

We recommend replacing magic constants with const variables with names.

Client's commentary

Fixed in debfd181.

It's been implemented using the recommended constants in the SATS contract.

Conclusion

Altogether, the audit process has revealed 1 HIGH, 3 MEDIUM, and 1 LOW severity finding.

Disclaimer

The Stronghold audit makes no statements or warranties about the utility of the code, the safety of the code, the suitability of the business model, investment advice, endorsement of the platform or its products, the regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug-free status. The audit documentation is for discussion purposes only.