BLOCKSEC

Security Audit
Report for Locked FBTC

Date: June 17, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Fire Bitcoin
Target	Locked FBTC

Version History

Version	Date	Description
1.0	June 17, 2024	First Release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-24 12:20:29+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is the Locked FBTC [1] of the Fire Bitcoin project. Locked FBTC is a companion contract for the FBTC and Fire Bridge contracts. The Locked FBTC has been locked within partner applications, backed by native BTC held in a separate BTC address. Locked FBTC is created for dedicated partner protocols. It is important to note that all dependencies of the smart contracts within the audit scope are considered reliable in both functionality and security. Therefore, they are excluded from the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Locked FBTC	`Version 1`	`ae3a6ace8e073115e1e82a338951c925bf8e3988`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

[1] https://github.com/fbtc-com/fbtcX-contract

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

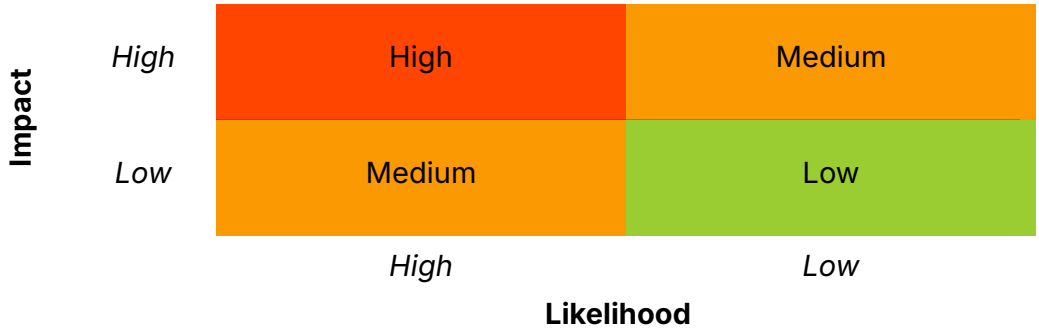
Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification



Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **one** note.

- Low Risk: 2
- Note: 1

ID	Severity	Description	Category	Status
1	Low	Potential forced partial redeem	DeFi Security	Acknowledged
2	Low	Potential race condition during the redeeming process	DeFi Security	Acknowledged
3	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential forced partial redeem

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description The `LockedFBTC` contract relies on the `FireBridge` contract to initiate bridging with the underlying (native) BTC when locking and unlocking the `LockedFBTC` tokens. For each bridging request, the `FireBridge` contract charges fees and returns the total amount of fees charged.

However, in the two-stage process of redeeming `FBTC` tokens from `LockedFBTC` tokens, the fees charged by the `FireBridge` contract are not considered. Specifically, in the following code segment, the amount of fees charged by the `FireBridge` contract is not used.

```
81  function redeemFbtcRequest(uint256 _amount, bytes32 _depositTxid, uint256 _outputIndex)
82      public
83      onlyRole(MINTER_ROLE)
84      whenNotPaused
85      returns (bytes32 _hash, Request memory _r)
86  {
87      require(_amount > 0 && _amount <= totalSupply(), "Amount out of limit.");
88
89      (_hash, _r) = IFireBridge(fbtcBridge).addMintRequest(_amount, _depositTxid, _outputIndex);
90      emit RedeemFbtcRequest(msg.sender, _depositTxid, _outputIndex, _amount);
91  }
```

Listing 2.1: LockedFBTC.sol

In the confirmation of the `FBTC` redemption, the amount of `FBTC` and `LockedFBTC` tokens are calculated as `1:1`, without accounting for any potential non-zero fees charged by the `FireBridge` contract.

```
93  function confirmRedeemFbtc(uint256 _amount) public onlyRole(MINTER_ROLE) whenNotPaused {
94      require(_amount > 0, "Amount must be greater than zero.");
```

```
95      require(fbtc.balanceOf(address(this)) >= _amount, "Insufficient FBTC balance in contract.")
            ;
96
97      _burn(msg.sender, _amount);
98      SafeERC20Upgradeable.safeTransfer(fbtc, msg.sender, _amount);
99
100     emit ConfirmRedeemFbtc(msg.sender, _amount);
101 }
```

Listing 2.2: LockedFBTC.sol

Impact Due to the fees charged during the redemption of `FBTC`, the previously minted `LockedFBTC` tokens cannot be fully redeemed.

Suggestion When processing redeem requests for `LockedFBTC`, consider the fees charged by `FireBridge`.

Feedback from the Project The project maintainers confirm that the fees charged when minting `FBTC` tokens in the `FireBridge` would be set to zero in the future.

2.1.2 Potential race condition during the redeeming process

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description The `LockedFBTC` contract uses a two-stage request-confirmation process for redeeming of `LockedFBTC` tokens. However, there is no internal accounting for the entire process, leading to a potential race condition where an account that requests later can confirm the redemption earlier.

Specifically, there is no internal accounting for the request process, as shown in the following code segment.

```
81  function redeemFbtcRequest(uint256 _amount, bytes32 _depositTxid, uint256 _outputIndex)
82      public
83      onlyRole(MINTER_ROLE)
84      whenNotPaused
85      returns (bytes32 _hash, Request memory _r)
86  {
87      require(_amount > 0 && _amount <= totalSupply(), "Amount out of limit.");
88
89      (_hash, _r) = IFireBridge(fbtcBridge).addMintRequest(_amount, _depositTxid, _outputIndex);
90      emit RedeemFbtcRequest(msg.sender, _depositTxid, _outputIndex, _amount);
91  }
```

Listing 2.3: LockedFBTC.sol

Furthermore, as shown in the following code segment, in the confirmation of the `FBTC` redemption, there are no checks for internal accounting. This allows any account with `LockedFBTC` tokens to redeem them whenever there is a sufficient balance in the `LockedFBTC` contract, bypassing the request process.

```
 93   function confirmRedeemFbtc(uint256 _amount) public onlyRole(MINTER_ROLE) whenNotPaused {
 94       require(_amount > 0, "Amount must be greater than zero.");
 95       require(fbtc.balanceOf(address(this)) >= _amount, "Insufficient FBTC balance in contract.")
              ;
 96
 97       _burn(msg.sender, _amount);
 98       SafeERC20Upgradeable.safeTransfer(fbtc, msg.sender, _amount);
 99
100       emit ConfirmRedeemFbtc(msg.sender, _amount);
101   }
```

Listing 2.4: LockedFBTC.sol

Impact There is a potential race condition if there are multiple accounts with the `MINTER_ROLE`.

Suggestion Refactor the redeem process of the `LockedFBTC` contract.

Feedback from the Project The project maintainers confirm that there will be multiple `LockedFBTC` contracts, each with only one account set as the `MINTER_ROLE`.

2.2 Note

2.2.1 Potential centralization risks

Description The `LockedFBTC` contract introduces mechanisms that may lead to centralization risks. Specifically, the `emergencyBurn` function can burn `LockedFBTC` tokens from any account with an arbitrary amount. This function is guarded by a privileged role, i.e., the `SAFETY_COMMITTEE_ROLE`.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS