Security Audit Report for STBT Contracts

Date: January 04, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.01.06 13:43:31 +08'00'

Contents

Report Manifest

Item	Description
Client	Matrixport
Target	STBT Contracts

Version History

Version	Date	Description
1.0	January 04, 2023	First Release

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 5 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 target of this audit is the STBT Contracts [1] of the Matrixport. Here *STBT* stands for Short-term Treasury Bond Token. It is a special token issued by the Matrixport team.

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
STBT Contracts	`Version 1`	`31227722a9ff42f48b8633c4f96c4086cb978478`
	`Version 2`	`187451c9d716f7bdf1507151f2688a83d2c56206`

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.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] `https://github.com/Matrixport-STBT/STBT-contracts`

- **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

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

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 find **one** potential issue. We also have **one** recommendation and **two** notes.

- Medium Risk: 1
- Recommendation: 1
- Note: 2

ID	Severity	Description	Category	Status
1	Medium	Potential precision loss in `getSharesByAmount`	Software Security	Acknowledged
2	-	Add sanity checks before setting parameters	Recommendation	Acknowledged
3	-	Permissioned token design	Note	-
4	-	External check for the TimeLock delays	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential precision loss in `getSharesByAmount`

Severity Medium

Status Acknowledged

Introduced by Version 1

Description In the `STBT` contract, there exists a potential precision loss issue. Specifically, the STBT token represents shares of the user. The amount of shares and the STBT token can be converted to each other. The share amount is calculated by the following formula (in the `getSharesByAmount` function, the `amount` stands for the amount of the STBT token):

$$shares = \frac{amount * totalShares}{totalSupply}$$

If `totalSupply` is zero, then the calculated value of `shares` is also zero.

```
function getSharesByAmount(uint256 _amount) public view returns (uint256 result) {
    // unchecked {
    //     result = _amount * totalShares / totalSupply; // divide-by-zero will return zero
    // }
    return totalSupply == 0 ? 0 : _amount * totalShares / totalSupply;
}
```

Listing 2.1: STBT.sol

While in the `issue` function, if the `amount` returned by the `getSharesByAmount` function is zero, the function would reset the contract state, i.e., the `totalSupply` would be reset to the `_value` at line 402. This is because the function assumes that the STBT token has not been minted if the return value of the `getSharesByAmount` function is zero.

```
395    function issue(address _tokenHolder, uint256 _value, bytes calldata _data) external onlyIssuer
         {
396        if (_value == 0) {
397            return;
```

```
    }
    uint amount = getSharesByAmount(_value);
    if (amount == 0) {
        amount = _value;
        totalSupply = _value;
        lastDistributeTime = uint64(block.timestamp);
    } else {
```

Listing 2.2: STBT.sol

Note that in the lifecycle of the STBT token, the `distributeInterests` function would be called to distribute interests to the users. This is done by increasing the `totalSupply`, so that each share would represent more STBT token. However, increasing the `totalSupply` implies the risk that the denominator gets larger in the `getSharesByAmount` function.

```
function distributeInterests(int256 _distributedInterest) external onlyIssuer {
    uint oldTotalSupply = totalSupply;
    uint newTotalSupply;
    if(_distributedInterest > 0) {
        require(oldTotalSupply * maxDistributeRatio >= uint(_distributedInterest) * (10 ** 18),
            'MAX_DISTRIBUTE_RATIO_EXCEEDED');
        newTotalSupply = oldTotalSupply + uint(_distributedInterest);
    } else {
        require(oldTotalSupply * maxDistributeRatio >= uint(-_distributedInterest) * (10 ** 18)
            , 'MAX_DISTRIBUTE_RATIO_EXCEEDED');
        newTotalSupply = oldTotalSupply - uint(-_distributedInterest);
    }
    totalSupply = newTotalSupply;
    require(lastDistributeTime + minDistributeInterval < block.timestamp, '
        MIN_DISTRIBUTE_INTERVAL_VIOLATED');
    emit InterestsDistributed(_distributedInterest, newTotalSupply, block.timestamp, block.
        timestamp - lastDistributeTime);
    lastDistributeTime = uint64(block.timestamp);
}
```

Listing 2.3: STBT.sol

As a result, if the `_value` is very small and the `totalSupply` is larger enough than the `totalShares`, the return value of the `getSharesByAmount` would be zero, thus resetting the `totalSupply` in the `issue` function.

Impact There is a chance that incorrect parameters and distributed interests can cause a precision loss in the `issue` function and reset the `totalSupply`.

Suggestion N/A

Feedback from the Project The project maintainers determines that the described precision loss is impossible because all the parameters in these functions are with 10^{18} precision, and the `issue` function is only callable by the issuer account.

2.2 Additional Recommendation

2.2.1 Add sanity checks before setting parameters

Status Acknowledged

Introduced by Version 1

Description It is recommended that to add proper checks before setting key parameters in the contract. For example, in the setMaxDistrbuteRatio function, only values larger than 10^{18} is meaningful for the setMaxDistrbuteRatio parameter.

```
function setMaxDistributeRatio(uint64 ratio) public onlyOwner {
    maxDistributeRatio = ratio;
}
```

Listing 2.4: STBT.sol

Impact N/A

Suggestion Implement proper checks before setting parameters.

Feedback from the Developers The proper range of the parameters can be hard to determine. If the range is too restrictive, it can limit the flexibility of the token. Otherwise, it is not useful to impose a range check.

2.3 Note

2.3.1 Permissioned token design

Description The STBT token uses a permissioned and centralized design that only permissioned accounts can transfer or receive STBT tokens. However, no accounts are permissioned in the permissions state variable by default.

```
    function _checkSendPermission(address _sender) private view {
        Permission memory permTx = permissions[_sender];
        require(permTx.sendAllowed, 'STBT: NO_SEND_PERMISSION');
        require(permTx.expiryTime == 0 || permTx.expiryTime > block.timestamp, 'STBT:
            SEND_PERMISSION_EXPIRED');
    }
    function _checkReceivePermission(address _recipient) private view {
        Permission memory permRx = permissions[_recipient];
        require(permRx.receiveAllowed, 'STBT: NO_RECEIVE_PERMISSION');
        require(permRx.expiryTime == 0 || permRx.expiryTime > block.timestamp, 'STBT:
            RECEIVE_PERMISSION_EXPIRED');
    }
```

Listing 2.5: STBT.sol

Feedback from the project The purpose of the STBT token (which is a Security Token), is that only users with permissions and proper KYC can participate the investment of the token.

2.3.2 External check for the TimeLock delays

Description In the StbtTimelockController contract, the implementation overwrites the minDelay mechanism in the original TimeLock contract. Specifically, it is replaced by setting separate delays for different selectors in the delayMap state variable in the constructor. However, the lowest limit for the delay is only 1 (second), which is enforced at line 52 in the following code snippet.

```
function schedule(
    address target,
    uint256 value,
    bytes calldata data,
    bytes32 predecessor,
    bytes32 salt,
    uint256 /*delay*/
) public override onlyRole(PROPOSER_ROLE) {

    bytes4 sel = bytes4(data[0:4]);
    uint256 delay = delayMap[sel];
    require(delay > 0, 'TimelockController: UNKNOWN_SELECTOR');

    super.schedule(target, value, data, predecessor, salt, delay);
}
```

Listing 2.6: STBT.sol

Feedback from the project There is an external check of the delays for the `StbtTimelockController` after the deployment. Only after the proper checks of the `StbtTimelockController`, would the `STBT` token contract be deployed.