Security Audit Report for Minter & WSTBT Contracts

Date: June 30, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by BlockSec
Audit Team
Date: 2023.06.30 11:56:46
+08'00'

Contents

Report Manifest

Item	Description
Client	Matrixdock
Target	Minter & WSTBT Contracts

Version History

Version	Date	Description
1.0	June 30, 2023	First Release

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 Minter & WSTBT Contracts [1] of the Matrixdock. It is important to note that the audit exclusively covers the following two contracts:`Minter.sol` and `WSTBT.sol`. The **WSTBT** contract is specifically designed to wrap or unwrap STBT tokens, while the **Minter** contract offers services that allow users to schedule mint and redeem operations, enabling the conversion between underlying and STBT tokens.

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
Minter & WSTBT Contracts	Version 1	0e4726b7195bb5adc12a664a5d2f4f1f7c80df62
	Version 2	68b4e34944695a20ce9f2e7acf833473f19c0ff4
	Version 3	b781aaf016538540792eadfbe5192006efe022dd
	Version 4	0f5d608eccf8a72c83b9e7e4682fae0cb7f5f828

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/Matrixport-STBT/STBT-contracts

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

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 **seven** potential issues. We also have **two** recommendations and **two** notes.

- Low Risk: 2
- Medium Risk: 2
- High Risk: 3
- Recommendation: 2
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Incorrect `nonceForMint` value emitted by the `Mint` event	DeFi Security	Fixed
2	Medium	Flawed verification of the `rescue` function	DeFi Security	Fixed
3	High	Improper conversion of `amount` in the `redeem` function	DeFi Security	Fixed
4	Low	Insufficient validation in the `mint` function	DeFi Security	Fixed
5	Medium	Lack of verification for the `token` in the `redeem` functions	DeFi Security	Fixed
6	High	Improper setting of `nonceForRedeemSettled` in the `redeemSettle` function	DeFi Security	Fixed
7	High	Potential DoS risk in preventing the invocation of the `rescue` function	DeFi Security	Fixed
8	-	Remove the redundant code in the `rescue` function	Recommendation	Fixed
9	-	Revise the incorrect comments	Recommendation	Acknowledged
10	-	Potential risks of uninitialized variables	Note	-
11	-	Centralization risk	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect `nonceForMint` value emitted by the `Mint` event

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `mint` function of the `Minter` contract, the current nonce (i.e., `nonceForMint`) is used to schedule the issuing operation and is subsequently incremented. The `Mint` event is then emitted to display the corresponding information for the mint. However, as demonstrated in lines 130 and 132 of Listing 2.1, the `Mint` event is emitted after updating the `nonceForMint` variable, leading to an incorrect value.

```
109 function mint(address token, uint depositAmount, uint minProposedAmount, bytes32 salt,
110         bytes calldata extraData) external {
111   {
112   (, bool receiveAllowed, uint64 expiryTime) = ISTBT(targetContract).permissions(msg.sender);
113   require(receiveAllowed, 'MINTER: NO_RECEIVE_PERMISSION');
```

```
    require(expiryTime == 0 || expiryTime > block.timestamp, 'MINTER: RECEIVE_PERMISSION_EXPIRED');
  }

  uint receiverAndRate = purchaseInfoMap.get(token);
  require(receiverAndRate != 0, "MINTER: TOKEN_NOT_SUPPORTED");
  address receiver = address(uint160(receiverAndRate>>96));
  uint feeRate = uint96(receiverAndRate);
  DepositConfig memory config = depositConfigMap[token];
  require(depositAmount >= config.minimumDepositAmount, "MINTER: DEPOSIT_AMOUNT_TOO_SMALL");
  uint proposeAmount = depositAmount*(UNIT-feeRate)/UNIT;
  proposeAmount = config.needDivAdjust? proposeAmount / config.adjustUnit : proposeAmount * config.adjustUnit;
  require(proposeAmount >= minProposedAmount, "MINTER: PROPOSE_AMOUNT_TOO_SMALL");
  IERC20(token).transferFrom(msg.sender, receiver, depositAmount);
  bytes memory data = abi.encodeWithSignature("issue(address,uint256,bytes)",
              msg.sender, proposeAmount, extraData);
  salt = keccak256(abi.encodePacked(salt, nonceForMint));
  nonceForMint = nonceForMint + 1;
  IStbtTimelockController(timeLockContract).schedule(targetContract, 0, data, bytes32(""), salt, 0);
  emit Mint(msg.sender, token, nonceForMint, depositAmount, proposeAmount, salt, data);
}
```

Listing 2.1: Minter.sol (in Version 1)

Impact The `Mint` event emits an incorrect `nonceForMint` value.

Suggestion Revise the code accordingly.

2.1.2 Flawed verification of the `rescue` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `rescue` function in the `Minter` contract is designed to recover users' accidentally sent assets if the last redemption is finalized (as shown in line 170 of Listing 2.2). However, the `redeemSettle` function lacks logic ensuring that redemptions are settled sequentially. Specifically, the owner could potentially settle the last redemption before settling prior ones. In other words, the rescue function can be activated even if there are unresolved redemptions. Consequently, the `rescue` function could be triggered with an incorrect token amount, including unprocessed or unredeemed tokens.

```
159 function redeemSettle(address token, uint amount, uint nonce, bytes32 redeemTxId,
160          uint redeemServiceFeeRate, uint executionPrice) onlyOwner external {
161   address target = redeemTargetMap[nonce];
162   require(target != address(0), "MINTER: NULL_TARGET");
163   IERC20(token).transfer(target, amount);
164   emit Settle(target, amount, redeemTxId, redeemServiceFeeRate, executionPrice);
165   delete redeemTargetMap[nonce];
166 }
167
```

```
// the rescue ETH or ERC20 tokens which were accidentally sent to this contract
function rescue(address token, address receiver, uint amount) onlyOwner external {
  require(redeemTargetMap[nonceForRedeem-1] == address(0), "MINTER: PENDING_REDEEM");
  if(token == address(0)) {
    (bool success,) = receiver.call{value : amount}(new bytes(0));
    require(success, "MINTER: FAIL_TO_RESCUE_ETH");
  } else {
    IERC20(token).transfer(receiver, amount);
  }
}
```

Listing 2.2: Minter.sol (in Version 1)

Impact The owner might be able to recover tokens associated with unprocessed redemptions.

Suggestion Verify and confirm that no redemptions exist in the `redeemTargetMap` before executing the rescue operation.

2.1.3 Improper conversion of `amount` in the `redeem` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the comments, the input `amount` for the `redeem` function represents the quantity of STBT tokens, while the input `token` stands for the underlying token. However, in line 142 of Listing 2.3, the variable `amount` is converted based on the configuration retrieved from `redeemConfigMap[token]` and is erroneously used to represent the STBT token quantity in lines 144 and 145.

```
135 // token: which token to receive after redeem?
136 // amount: how much STBT to deposit?
137 // salt: a random number that can affect TimelockController's input salt
138 // extraData: will be used to call STBT's issue functions
139 function redeem(uint amount, address token, bytes32 salt, bytes calldata extraData) external {
140   RedeemConfig memory config = redeemConfigMap[token];
141   require(amount >= config.minimumRedeemAmount, "MINTER: REDEEM_AMOUNT_TOO_SMALL");
142   amount = config.needDivAdjust? amount / config.adjustUnit : amount * config.adjustUnit;
143   bytes memory data = abi.encodeWithSignature("redeemFrom(address,uint256,bytes)",
144                 poolAccount, amount, extraData);
145   IERC20(targetContract).transferFrom(msg.sender, poolAccount, amount);
146   salt = keccak256(abi.encodePacked(salt, nonceForRedeem));
147   IStbtTimelockController(timeLockContract).schedule(targetContract, 0, data, bytes32(""), salt,
          0);
148   redeemTargetMap[nonceForRedeem] = msg.sender;
149   emit Redeem(msg.sender, token, nonceForRedeem, amount, salt, data);
150   nonceForRedeem = nonceForRedeem + 1;
151 }
```

Listing 2.3: Minter.sol (in Version 1)

Consequently, an incorrect quantity of STBT tokens is transferred, which goes against the users' intentions. On the other hand, if the input `amount` for the `redeem` function signifies the quantity of the input token, the Redeem event is emitted with a converted amount that does not accurately represent the input token quantity.

Impact An incorrect quantity of STBT tokens may be transferred, contrary to users' intentions.

Suggestion Implement a new variable to store the converted input amount value and use it accordingly.

2.1.4 Insufficient validation in the mint function

Severity Low

Status Fixed in Version 3

Introduced by Version 1

Description The mint function in the Minter contract does not validate the input token, depositAmount, and minProposedAmount. Consequently, if users invoke the mint function with an unset token (in the depositConfigMap mapping), zero depositAmount, and zero minProposedAmount, they can pass all validations in the mint function (lines 123 and 126 of Listing 2.4) and successfully schedule a mint operation without depositing any underlying tokens. As a result, malicious users can initiate multiple zero-mint operations to increase the burden of processing these non-meaningful mint operations.

```
110  function mint(address token, uint depositAmount, uint minProposedAmount, bytes32 salt,
111         bytes calldata extraData) external {
112    {
113    (, bool receiveAllowed, uint64 expiryTime) = ISTBT(targetContract).permissions(msg.sender);
114    require(receiveAllowed, 'MINTER: NO_RECEIVE_PERMISSION');
115    require(expiryTime == 0 || expiryTime > block.timestamp, 'MINTER: RECEIVE_PERMISSION_EXPIRED')
           ;
116    }
117
118    uint receiverAndRate = purchaseInfoMap.get(token);
119    require(receiverAndRate != 0, "MINTER: TOKEN_NOT_SUPPORTED");
120    address receiver = address(uint160(receiverAndRate>>96));
121    uint feeRate = uint96(receiverAndRate);
122    DepositConfig memory config = depositConfigMap[token];
123    require(depositAmount >= config.minimumDepositAmount, "MINTER: DEPOSIT_AMOUNT_TOO_SMALL");
124    uint proposeAmount = depositAmount*(UNIT-feeRate)/UNIT;
125    proposeAmount = config.needDivAdjust? proposeAmount / config.adjustUnit : proposeAmount *
           config.adjustUnit;
126    require(proposeAmount >= minProposedAmount, "MINTER: PROPOSE_AMOUNT_TOO_SMALL");
127    IERC20(token).transferFrom(msg.sender, receiver, depositAmount);
128    bytes memory data = abi.encodeWithSignature("issue(address,uint256,bytes)",
129                msg.sender, proposeAmount, extraData);
130    uint _nonceForMint = nonceForMint;
131    salt = keccak256(abi.encodePacked(salt, _nonceForMint));
132    nonceForMint = nonceForMint + 1;
133    IStbtTimelockController(timeLockContract).schedule(targetContract, 0, data, bytes32(""), salt,
            0);
134    emit Mint(msg.sender, token, _nonceForMint, depositAmount, proposeAmount, salt, data);
135  }
```

Listing 2.4: Minter.sol (in Version 1)

Impact Increase the processing burden for zero-mint operations.

Suggestion Verify the input `token` and ensure that the token is set in the `depositConfigMap` variable.

2.1.5 Lack of verification for the input `token` in the `redeem` functions

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The `redeem` function in the `Minter` contract enables users to redeem their underlying tokens by sending STBT tokens. However, the redeem function does not validate the input `token`. As a result, if users invoke the `redeem` function with a non-existing `token` (which is unset in the `redeemConfigMap` mapping) and zero `amount`, they can pass the validations in the `redeem` function (line 143 of Listing 2.5) and successfully schedule a mint operation without transferring any STBT tokens. Furthermore, the `nonceForRedeem` variable will be incremented at the end of the `redeem` function.

```
141  function redeem(uint amount, address token, bytes32 salt, bytes calldata extraData) external {
142    RedeemConfig memory config = redeemConfigMap[token];
143    require(amount >= config.minimumRedeemAmount, "MINTER: REDEEM_AMOUNT_TOO_SMALL");
144    IERC20(targetContract).transferFrom(msg.sender, poolAccount, amount);
145    bytes memory data = abi.encodeWithSignature("redeemFrom(address,uint256,bytes)",
146                 poolAccount, amount, extraData);
147    uint adjusted = config.needDivAdjust? amount / config.adjustUnit : amount * config.adjustUnit;
148    salt = keccak256(abi.encodePacked(salt, nonceForRedeem));
149    IStbtTimelockController(timeLockContract).schedule(targetContract, 0, data, bytes32(""), salt,
         0);
150    redeemTargetMap[nonceForRedeem] = msg.sender;
151    emit Redeem(msg.sender, token, nonceForRedeem, adjusted, salt, data);
152    nonceForRedeem = nonceForRedeem + 1;
153  }
```

Listing 2.5: Minter.sol (in Version 2)

According to the design of the redeem process, the owner of the `Minter` contract will invoke the `redeemSettle` function to return redeemed underlying tokens sequentially based on the `nonceForRedeemSettled` variable. Consequently, malicious users can schedule multiple redeem operations with non-existing tokens and zero amounts to obstruct the redeem process. Additionally, the rescue process can be affected since it can only be triggered when there are no pending redeem operations. This situation can lead to potential DoS issues that block the redeem and rescue process.

Impact Potentially cause DoS issues to block the redeem and rescue process.

Suggestion Verify the input `token` within the `redeem` function.

2.1.6 Improper setting of `nonceForRedeemSettled` in the `redeemSettle` function

Severity High

Status Fixed in `Version 4`

Introduced by `Version 2`

Description The `redeemSettle` function in the Minter contract returns redeemed underlying tokens to users and updates the `nonceForRedeemSettled` variable. Furthermore, based on the `nonceForRedeem` variable, the `redeemSettle` function processes redeem operations sequentially. To ensure that redeem operations are settled in sequence, the first require verification (line 163 of Listing 2.6) checks if the provided nonce is equal to `nonceForRedeemSettled + 1`. However, both `nonceForRedeem` and `nonceForRedeemSettled` variables are uninitialized and start with the default value of 0. Consequently, the owner of the Minter contract cannot successfully trigger the `redeemSettle` function with a nonce of 0, which leads to users not being refunded.

```
161  function redeemSettle(address token, uint amount, uint64 nonce, bytes32 redeemTxId,
162           uint redeemServiceFeeRate, uint executionPrice) onlyOwner external {
163    require(nonce == nonceForRedeemSettled + 1, "MINTER: INVALID_NONCE");
164    nonceForRedeemSettled = nonce;
165    address target = redeemTargetMap[nonce];
166    require(target != address(0), "MINTER: NULL_TARGET");
167    IERC20(token).transfer(target, amount);
168    emit Settle(target, amount, redeemTxId, redeemServiceFeeRate, executionPrice);
169    delete redeemTargetMap[nonce];
170  }
```

Listing 2.6: Minter.sol (in Version 2)

Impact The `redeemSettle` function can never be triggered to refund underlying tokens to users.

Suggestion Initialize the `nonceForRedeem` variable with a value of 1

2.1.7 Potential DoS risk in preventing the invocation of the `rescue` function

Severity High

Status Fixed in `Version 4`

Introduced by `Version 2`

Description The `rescue` function in the `Minter` contract is designed to recover users' accidentally sent assets if the last redemption is finalized (as shown in line 174 of Listing 2.7). However, the `require` verification in the `rescue` function is flawed. Specifically, the variable `nonceForRedeemSettled` records the last settled redemption, and the variable `nonceForRedeem` records the nonce for the next redemption request. This means that the variables `nonceForRedeemSettled` and `nonceForRedeem` can never be equal. As a result, the `require` verification of the `rescue` function cannot be passed, leading to the failure of the `rescue` function.

```
173  function rescue(address token, address receiver, uint amount) onlyOwner external {
174    require(nonceForRedeemSettled == nonceForRedeem, "MINTER: PENDING_REDEEM");
175    IERC20(token).transfer(receiver, amount);
176  }
```

Listing 2.7: Minter.sol (in Version 2)

Impact Potential DoS risk in obstructing the redeem and rescue process.

Suggestion Verify the input `token` within the `redeem` function.

2.2 Additional Recommendation

2.2.1 Remove the redundant code in the `rescue` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `rescue` function of the `Minter` contract, a low-level call (line 172 in the following code snippet) is utilized to rescue the contract's Ethers and transfer them to the receiver's address. However, this contract does not have any `payable` functions, meaning it cannot receive Ethers from any source. As a result, the first part of the `rescue` function (lines 171-173 in Listing 2.8) appears to be ineffective.

```
169  function rescue(address token, address receiver, uint amount) onlyOwner external {
170    require(redeemTargetMap[nonceForRedeem-1] == address(0), "MINTER: PENDING_REDEEM");
171    if(token == address(0)) {
172      (bool success,) = receiver.call{value : amount}(new bytes(0));
173      require(success, "MINTER: FAIL_TO_RESCUE_ETH");
174    } else {
175      IERC20(token).transfer(receiver, amount);
176    }
177  }
```

Listing 2.8: Minter.sol (in Version 1)

Impact N/A

Suggestion Remove the redundant code.

2.2.2 Revise the incorrect comments

Status Acknowledged

Introduced by `Version 1`

Description It is recommended to review the comments within the `Minter` contract to prevent any misunderstandings. For example, the `redeem` and `mint` functions have identical explanations for the `extraData` variable, which may not be correct.

Impact N/A

Suggestion Remove the redundant code.

2.3 Note

2.3.1 Potential risks of uninitialized variables

Description In the `STBT` contract, some variables like `lastDistributeTime`, `minDistributeInterval`, and `maxDistributeRatio` are not initialized before assignments or other usages. This could potentially introduce risks.

Feedback from the project Prior to the official launch of the service, these three variables will be initialized by invoking the `setMinDistributeInterval`, `setMaxDistributeRatio`, and `issue` functions, respectively.

2.3.2 Centralization risk

Description The two contracts contain several privileged functions, such as the `redeemSettle` and `rescue` functions in the `Minter` contract, which can only be triggered by the owner to transfer assets from the contract. This introduces a centralization risk, as privileged accounts have control over the assets.