BLOCKSEC

Security Audit Report for Token Contracts

Date: June 7, 2024 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Particle
Target	Token Contracts

Version History

Version	Date	Description
1.0	June 7, 2024	First Release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-07 08:59:53+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 on the Token Contracts of Particle [1]. These Token Contracts integrate core Particle token logic, allowing users to stake Particle tokens and earn airdrops. It is important to note that the `TOKEN` in the core contracts is assumed to be the Particle token and cannot be modified once deployed. For design considerations, the Particle token contract is unupgradeable.

Only the contracts located within the `contracts` and `script` folders in the repository are included in the scope of this audit. Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and therefore, they are not included in the scope of the audit.

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
Token Contracts	`Version 1`	`1f60831bb66fdaae243396cdebc2a549bd2300c8`
	`Version 2`	`576825234d75fe1b3166df161fb365cf9003fe6e`

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.

[1] https://github.com/Particle-Platforms/token-contract

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.

- **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.

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

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

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

Overall, we did not find potential security issues. However, there are **three** recommendations and **three** notes.

- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	-	Add checks on key parameters	Recommendation	Fixed
2	-	Follow CEI pattern in the `deposit` function	Recommendation	Fixed
3	-	Emit events for key functions	Recommendation	Fixed
4	-	Potential front-running risks	Note	-
5	-	Potential centralization risks	Note	-
6	-	Reward distribution are controlled by off-chain logic	Note	-

The details are provided in the following sections.

2.1 Additional Recommendation

2.1.1 Add checks on key parameters

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Several functions lack checks on key parameters. For instance, the constructor of the `Airdrop` contract does not ensure that `TOKEN` is a non-zero address, and the `setManager` function does not impose checks on the new manager address. Since misconfiguration with a zero address would be irreversible, it is recommended to add checks to ensure that the provided addresses are valid and non-zero.

```
30    constructor(address token) Ownable(msg.sender) {
31        TOKEN = token;
32    }
```

Listing 2.1: contracts/Airdrop.sol

```
33    function setManager(address _manager) external onlyManager {
34        manager = _manager;
35    }
```

Listing 2.2: contracts/libraries/BlastManager.sol

Impact The contracts may not function properly due to potential misconfigurations.

Suggestion Add sanity checks on key parameters.

2.1.2 Follow CEI pattern in the `deposit` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `deposit` function, the updates to the `stakes` mapping occur before the external call to transfer users' tokens in. This violates the common CEI (Checks-Effects-Interactions) programming pattern, which dictates that the state variable should be updated before conducting the external call. It is recommended to follow the CEI pattern as implemented in the `withdraw` function.

```
59  function deposit(uint256 amount) external nonReentrant {
60      IERC20(TOKEN).safeTransferFrom(msg.sender, address(this), amount);
61
62      _syncStake(msg.sender);
63      stakes[msg.sender].principal += amount;
64
65      emit Deposit(msg.sender, amount);
66  }
```

Listing 2.3: contracts/Staking.sol

```
73  function withdraw(address to, uint256 amount) external nonReentrant {
74      Stake memory stake = stakes[msg.sender];
75
76      require(stake.principal >= amount, "Staking: insufficient balance");
77
78      _syncStake(msg.sender);
79      stakes[msg.sender].principal -= amount;
80
81      IERC20(TOKEN).safeTransfer(to, amount);
82
83      emit Withdraw(msg.sender, to, amount);
84  }
```

Listing 2.4: contracts/Staking.sol

Impact N/A

Suggestion Update the `stakes` before the `safeTransferFrom` call.

2.1.3 Emit events for key functions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In Token Contracts, several privileged functions lack event emissions. For instance, the `withdraw` function in the `Lockup` contract allows the owner, which should be the `Timelock` contract, to transfer an arbitray amount of tokens but doe not emit events. It is recommended that these privileged functions emit events to facilitate off-chain monitoring of critical operations.

```
28    function withdraw(address to, uint256 amount) external onlyOwner nonReentrant {
29        IERC20(TOKEN).safeTransfer(to, amount);
30    }
```

Listing 2.5: contracts/Lockup.sol

Impact N/A

Suggestion Emit events for key functions.

2.2 Note

2.2.1 Potential front-running risks

Introduced by `Version 1`

Description In the `Airdrop` contract, the `updateMerkleRoot` and `setStartTime` functions set the Merkle root and start time for an airdrop, respectively. The `onlyStarted` modifier ensures that users can only claim airdrops after the start time. However, the contract may be vulnerable to a potential front-run attack. If the calls to `updateMerkleRoot` and `setStartTime` are divided into two separate transactions, malicious users could front-run the second transaction with their own `claim` transactions. Since the `startTime` is not updated yet, they could effectively bypass the `onlyStarted` check and claim airdrops before the intended start time.

```
34    /**
35     * notice Update the merkle root for the airdrop* param root The new merkle root
36     */
37    function updateMerkleRoot(bytes32 root) external onlyOwner {
38        _merkleRoot = root;
39    }
40
41    /**
42     * notice Set the start time for the airdrop* param _startTime The new start time
43     */
44    function setStartTime(uint256 _startTime) external onlyOwner {
45        startTime = _startTime;
46    }
```

Listing 2.6: contracts/Airdrop.sol

```
67    modifier onlyStarted() {
68        if (startTime > 0) {
69            require(block.timestamp >= startTime, "Airdrop: not started");
70        }
71        _;
72    }
```

Listing 2.7: contracts/Airdrop.sol

Feedback from the Project To mitigate the risk of potential front-running attacks, the team ensures that the `startTime` is updated before setting the `merkleRoot`.

2.2.2 Potential centralization risks

Introduced by Version 1

Description The protocol carries potential centralization risks. Specifically, the privileged roles `owner` and `manager` have the ability to modify some key configurations, such as the Merkle root and start time for airdrops. Additionally, the `owner` of the `Lockup` contract is granted the ability to transfer an arbitrary amount of Particle tokens.

During this audit, the `Lockup`'s `owner` is assumed to be the `Timelock` contract, which inherits OpenZepplin's `TimelockController` and should be correctly deployed. The ownership of the other contracts is assigned to a multi-signature wallet. If (most of) the private keys of this wallet are compromised, it could lead to losses for the entire protocol.

2.2.3 Reward distribution are controlled by off-chain logic

Introduced by Version 1

Description The `staking` contract tracks a `principalTimespan` for each user based on their staking amount and timespan but is not used in the current code implementation. The staking reward calculation and distribution are currently controlled by off-chain logic, and thus fall outside the scope of this audit.

Feedback from the Project Currently, staking involves simply recording each user's staking amount multiplied by the timespan. This allows staking rewards to be recorded off-chain, and future contracts can read this value to enable users to receive tokens. In the later stages, there will be a new airdrop contract deployment, and a multi-sig will transfer tokens from the lockup to the new airdrop through the timelock proposal. Then, the airdrop will allow users to claim tokens based on their staking value (without using the Merkle tree). However, this process is reserved for later stages and is not part of the current deployment.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS