Security Audit Report for Minted Reward

Date: August 05, 2022

Version: 1.1

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2022.08.05 15:56:43
+08'00'

Contents

Report Manifest

Item	Description
Client	Minted
Target	Minted Reward

Version History

Version	Date	Description
1.1	August 05, 2022	Add feedback from the project
1.0	July 28, 2022	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

1.2 Executive Summary

The audit was performed by two auditors independently. From July 18th to July 26th, two auditors first leveraged the in-house static analysis tool to fully scan the code for known vulnerabilities. For each reported issue, the auditors manually checked whether it's a false alarm or a true positive. Besides, the auditors read the code to fully understand the business logic of the contract and the interactions between different contracts. They then enumerate possible attack surfaces and write PoC code to check whether the attack surface really exists.

Moreover, the auditing process is iterative. The initial version and following commits that fix the discovered issues were audited. If there are new issues, this process will continue.

Project	Version	File Name	File MD5
Minted Reward	Version 1	MintedAirdrop.sol	f01d27468bbccb235a0a23138e2384a1
		RewardController.sol	c1735364972d2cc3f648ad35d23aa199
		MintedToken.sol	e798f89d8dc8295ab343c5f68b14da25
		RewardDistributorWithVesting.sol	47d9a77366b606804a8116ea4064d5f6
		FeeDistributor.sol	f1744ab9e1d23843ef0c11a71cba2e27
		TokenDistributor.sol	dc8805a1607a14a00494a733dced7359
		TokenSplitter.sol	5cc3434509a4c87a008a63abc2956847
		MintedBoost.sol	5fe3ed4ac4da7124ffcb60adcc89b397
		MintedVesting.sol	4546e7b26fa1b7a1a42852bbb25ec742
		RewardsDistributor.sol	2a3530ddc18b73ea36f11600b8682e61
	Version 2	MintedAirdrop.sol	f01d27468bbccb235a0a23138e2384a1
		RewardController.sol	c1735364972d2cc3f648ad35d23aa199
		MintedToken.sol	e798f89d8dc8295ab343c5f68b14da25
		RewardDistributorWithVesting.sol	5adaa560558590860603b4c56f726353
		FeeDistributor.sol	f1744ab9e1d23843ef0c11a71cba2e27
		TokenDistributor.sol	5c367d3560768a453044d3193f7e7c1b
		TokenSplitter.sol	95ee6708a01df1dde2f907b45897d237
		MintedBoost.sol	5fe3ed4ac4da7124ffcb60adcc89b397
		MintedVesting.sol	18f1e1ac940e18366538fef0a3d29e23
		RewardsDistributor.sol	e0de02573d4a54745b7c59bc2cb7ccbd

1.3 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 do 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.

1.4 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.

1.4.1 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

1.4.2 DeFi Security

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

1.4.3 NFT Security

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

1.4.4 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.*

1.5 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 [1] and Common Weakness Enumeration [2]. 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.

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 issue will fall into one of the following four categories:

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

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

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

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

- **Fixed** The issue has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **five** potential issues. We have **five** recommendations.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 5
- Recommendations: 5
- Notes: 0

ID	Severity	Description	Category	Status
1	Low	Duplicate checks	Software Security	Fixed
2	Low	Lack of checks to enforce the validity of import variables	DeFi Security	Fixed
3	Low	Potential inconsistency caused by invoking the `updateShareDistribution()` function	DeFi Security	Fixed
4	Low	Inconsistent operation when invoking the `_pause()` function	DeFi Security	Confirmed
5	Low	The staked token's `unlockTimestamp` is not updated with new `_lockPeriod`	DeFi Security	Confirmed
6	-	Add non-zero address check	Recommendation	Fixed
7	-	Perform early check to save gas	Recommendation	Fixed
8	-	Avoid unnecessary calculation	Recommendation	Fixed
9	-	Avoid duplicate calculation	Recommendation	Confirmed
10	-	Be aware of that the `getPendingReward()` function may return inaccurate result	Recommendation	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Duplicate checks

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Checks in line 82 and line 83 are exactly the same. According to the context, one of them should be `_rewardsPerBlockForOthers`.

```
70    constructor(
71        address _mintedToken,
72        address _tokenSplitter,
73        uint256 _startBlock,
74        uint256[] memory _rewardsPerBlockForStaking,
75        uint256[] memory _rewardsPerBlockForOthers,
76        uint256[] memory _periodLengthesInBlocks,
77        uint256 _numberPeriods,
78        uint256 slippage
```

```
79    ) {
80        require(
81            (_periodLengthesInBlocks.length == _numberPeriods) &&
82                (_rewardsPerBlockForStaking.length == _numberPeriods) &&
83                (_rewardsPerBlockForStaking.length == _numberPeriods),
84            "Distributor: lengths must match numberPeriods"
85        );
86        require(_tokenSplitter != address(0), "Distributor: tokenSplitter must not be address(0)");
87        ...
```

Listing 2.1: TokenDistributor.sol

Impact N/A

Suggestion Revise the code accordingly.

2.2 DeFi Security

2.2.1 Lack of checks to enforce the validity of import variables

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `RewardsDistributor.sol`, the variable named `maximumAmountPerUserInCurrentTree` can only be assigned in the `updateTradingRewards()` function. For a given merkleRoot of the current reward round, this value cannot be changed.

```
90    function updateTradingRewards(bytes32 merkleRoot, uint256 newMaximumAmountPerUser) external
          onlyOwner {
91        require(!merkleRootUsed[merkleRoot], "Owner: Merkle root already used");
92        currentRewardRound++;
93        merkleRootOfRewardRound[currentRewardRound] = merkleRoot;
94        merkleRootUsed[merkleRoot] = true;
95        maximumAmountPerUserInCurrentTree = newMaximumAmountPerUser;
96        emit UpdateTradingRewards(currentRewardRound);
97    }
```

Listing 2.2: RewardsDistributor.sol

Meanwhile, this variable is used in the `claim()` function to verify the amount. A mistaken value might break the use of the `claim()` function.

```
63    function claim(uint256 amount, bytes32[] calldata merkleProof) external whenNotPaused
          nonReentrant {
64        // Verify the reward round is not claimed already
65        require(!hasUserClaimedForRewardRound[currentRewardRound][msg.sender], "Rewards: Already
              claimed");
66        require(amount >= amountClaimedByUser[msg.sender], "Rewards: underflow claim amount");
67
68        (bool claimStatus, uint256 adjustedAmount) = _canClaim(msg.sender, amount, merkleProof);
69        require(claimStatus, "Rewards: Invalid proof");
70        require(maximumAmountPerUserInCurrentTree >= amount, "Rewards: Amount higher than max");
```

```
71
72          // Set mapping for user and round as true
73          hasUserClaimedForRewardRound[currentRewardRound][msg.sender] = true;
74
75          // Adjust amount claimed
76          amountClaimedByUser[msg.sender] += adjustedAmount;
77          totalClaimed += adjustedAmount;
78
79          // Transfer adjusted amount
80          mintedToken.safeTransfer(msg.sender, adjustedAmount);
81
82          emit RewardsClaim(msg.sender, currentRewardRound, adjustedAmount);
83      }
```

Listing 2.3: RewardsDistributor.sol

In `RewardDistributorWithVesting.sol`, the variable named `maximumAmountPerUserInCurrentTree` has the similar issue.

Impact The mistaken values might break the use of the corresponding functions.

Suggestion Add proper sanity checks.

2.2.2 Potential inconsistency caused by invoking the `updateShareDistribution()` function

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `RewardDistributorWithVesting.sol`, there exists a potential inconsistency can be caused by invoking the `updateShareDistribution()` function. Specifically, suppose in the `constructor`, the variable `rewardSharesToUser` is specified as 70, while the variable `rewardSharesToStake` is set to 30, e.g., A (15), B (10) and C (5), respectively.

```
74      constructor(
75          address _mintedToken,
76          uint256 _rewardSharesToUser,
77          uint256 _rewardSharesToStake,
78          uint256[] memory _pidsToStake,
79          uint256[] memory _stakeshares
80      ) {
81          _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
82          _grantRole(UPDATE_REWARD_ROLE, msg.sender);
83          require(_pidsToStake.length == _stakeshares.length, "length mismatch");
84          require(_rewardSharesToUser + _rewardSharesToStake == 100_00, "wrong reward shares
                distribution");
85          rewardSharesToUser = _rewardSharesToUser;
86          for (uint256 i; i < _pidsToStake.length; i++) {
87              require(stakePoolInfo[_pidsToStake[i]].active == false, "duplicate pid");
88              rewardSharesToStake += _stakeshares[i];
89              stakePoolInfo[_pidsToStake[i]] = StakePool(_stakeshares[i], true);
90              stakedPids.push(_pidsToStake[i]);
91          }
```

```
92
93        require(rewardSharesToStake == _rewardSharesToStake, "wrong stake shares distribution");
94        mintedToken = IERC20(_mintedToken);
95        _pause();
96    }
```

Listing 2.4: RewardDistributorWithVesting.sol

Then the arguments to invoke the `updateShareDistribution()` function can be specified as A (15) and B (15). Then all the checks can be bypassed, and we have A (15), B (15) and C(5). As a result, although `rewardSharesToStake` is 30, the total share stored in `stakePoolInfo` is 35.

```
131   function updateShareDistribution(
132       uint256 _rewardSharesToUser,
133       uint256 _rewardSharesToStake,
134       uint256[] memory _pidsToStake,
135       uint256[] memory _stakeshares
136   ) external onlyRole(DEFAULT_ADMIN_ROLE) {
137       require(address(mintedBoost) != address(0), "mintedBoost is not setup");
138       require(_pidsToStake.length == _stakeshares.length, "length mismatch");
139       require(_rewardSharesToUser + _rewardSharesToStake == 100_00, "wrong reward shares
              distribution");
140
141       rewardSharesToUser = _rewardSharesToUser;
142       uint256 totalStakeShares;
143       for (uint256 i; i < _pidsToStake.length; i++) {
144           require(stakePoolInfo[_pidsToStake[i]].active == true, "pid doesn't exist");
145           totalStakeShares += _stakeshares[i];
146           stakePoolInfo[_pidsToStake[i]].shares = _stakeshares[i];
147       }
148       require(totalStakeShares == _rewardSharesToStake, "wrong stake shares distribution");
149       rewardSharesToStake = _rewardSharesToStake;
150
151       emit UpdateShareDistribution(_rewardSharesToUser, _rewardSharesToStake);
152   }
```

Listing 2.5: RewardDistributorWithVesting.sol

Therefore the arguments of the `updateShareDistribution()` function need to be checked to avoid such an inconsistency. Note that a similar issue also exists in the `updateShareDistribution()` function of `TokenSplitter.sol`.

Impact N/A

Suggestion Add proper sanity checks.

2.2.3 Inconsistent operation when invoking the `_pause()` function

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In `RewardsDistributor.sol`, the inconsistent operation when invoking the `_pause()` function may bypass the **BUFFER TIME** mechanism of this project. Specifically, the variable named `lastPausedTimestamp` will be assigned each time the `_pause()` function is invoked.

```
53    constructor(address _mintedToken) {
54        mintedToken = IERC20(_mintedToken);
55        _pause();
56    }
```

Listing 2.6: RewardsDistributor.sol

However, it is not specified the value in the `constructor` which also contains an invocation of the `_pause()` function.

```
102   function pauseDistribution() external onlyOwner whenNotPaused {
103       lastPausedTimestamp = block.timestamp;
104       _pause();
105   }
```

Listing 2.7: RewardsDistributor.sol

The similar issue also occurs in `RewardDistributorWithVesting.sol` and `MintedAirdrop.sol`.

Impact This issue may bypass the BUFFER TIME mechanism.

Suggestion N/A

Communication with the Project

The Developer: Acknowledge that the owner can withdraw the mtd token rewards without delay if the owner does not unpauseDistribution(). In our deployment setup, we'll unpauseDistribution() before the reward starts streaming into the contract which should mitigate this concern.

The Auditor: Thanks for the information. I think whether need to fix this issue depends on the design. Since there is a buffer time for withdrawing token rewards in other functions, we thought it needs a buffer time for each withdrawal. However, if this is the intended design for not having the buffer time during the setup time, it's fine not to fix this.

2.2.4 The staked token's `unlockTimestamp` is not updated with new `_lockPeriod`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description If the new `_lockPeriod` is different from the old one, then the staked token in this pool should update their `unlockTimestamp`.

```
330   function set(
331       uint256 _pid,
332       uint256 _multiplier,
333       uint256 _lockPeriod
334   ) public onlyOwner {
335       require(activePoolMap[_multiplier][_lockPeriod] == 0, "MintedBoost: Duplicate Pool");
336       require(_multiplier > 0, "MintedBoost: Multiplier must be > 0");
337       _harvest();
338
```

```
339         PoolInfo storage pool = poolInfo[_pid];
340         activePoolMap[pool.multiplier][pool.lockPeriod] = 0;
341         pool.multiplier = _multiplier;
342         pool.lockPeriod = _lockPeriod;
343         activePoolMap[_multiplier][_lockPeriod] = _pid + 1;
344
345         emit SetPool(_pid, _multiplier, _lockPeriod);
346     }
```

Listing 2.8: MintedBoost.sol

Impact N/A

Suggestion N/A

Feedback from the Project This is our product design. We do not want to impact the user's existing stake if we change the multiplier or lock period.

2.3 Additional Recommendation

2.3.1 Add non-zero address check

Status Fixed in `Version 2`

Introduced by `Version 1`

Description It is recommended to add more non-zero address checks, e.g., the first argument in the constructor of TokenDistribution.sol.

```
70      constructor(
71          address _mintedToken,
72          address _tokenSplitter,
73          uint256 _startBlock,
74          uint256[] memory _rewardsPerBlockForStaking,
75          uint256[] memory _rewardsPerBlockForOthers,
76          uint256[] memory _periodLengthesInBlocks,
77          uint256 _numberPeriods,
78          uint256 slippage
79      ) {
80          require(
81              (_periodLengthesInBlocks.length == _numberPeriods) &&
82                  (_rewardsPerBlockForStaking.length == _numberPeriods) &&
83                  (_rewardsPerBlockForStaking.length == _numberPeriods),
84              "Distributor: lengths must match numberPeriods"
85          );
86          require(_tokenSplitter != address(0), "Distributor: tokenSplitter must not be address(0)");
87
88          // 1. Operational checks for supply
89          uint256 nonCirculatingSupply = IMintedToken(_mintedToken).SUPPLY_CAP() -
90              IMintedToken(_mintedToken).totalSupply();
91
92          uint256 amountTokensToBeMinted;
93          ...
```

Listing 2.9: ERC20RootVault.sol

Impact N/A

Suggestion Add the sanity checks.

2.3.2 Perform early check to save gas

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `RewardsDistributor.sol` (and `RewardDistributorWithVesting.sol`), the check can be moved to the beginning of the `_canClaim()` function to avoid unnecessary computation.

```
149    function _canClaim(
150        address user,
151        uint256 amount,
152        bytes32[] calldata merkleProof
153    ) internal view returns (bool, uint256) {
154        // Compute the node and verify the merkle proof
155        bytes32 node = keccak256(abi.encodePacked(user, amount));
156        bool canUserClaim = MerkleProof.verify(
157            merkleProof,
158            merkleRootOfRewardRound[currentRewardRound],
159            node
160        );
161        if (
162            (!canUserClaim) ||
163            (hasUserClaimedForRewardRound[currentRewardRound][user]) ||
164            amount < amountClaimedByUser[user]
165        ) {
166            return (false, 0);
167        } else {
168            return (true, amount - amountClaimedByUser[user]);
169        }
170    }
```

Listing 2.10: RewardsDistributor.sol

Impact N/A

Suggestion Revise the code accordingly.

2.3.3 Avoid unnecessary calculation

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In line 155, `vestedIntervals * vesting.interval` is used to perform the calculation. However, `vestedIntervals` is specified as `timeFromStart / vesting.interval` in line 152. Obviously, it is an unnecessary calculation because `timeFromStart` could be used in line 155 directly.

```
140    function calculateVestingAmount(address _receiver) public view returns (uint256) {
141        Vesting memory vesting = vestingInfo[_receiver];
142        if (block.timestamp < (vesting.startTime + vesting.cliff)) {
143            return 0;
```

```
144         }
145         uint256 timeFromStart = block.timestamp - vesting.startTime;
146 
147         uint256 availableAmount = 0;
148         if (timeFromStart >= vesting.duration) {
149             availableAmount = vesting.allocated - vesting.released;
150         } else {
151             // Calculate number of intervals so far
152             uint256 vestedIntervals = timeFromStart / vesting.interval;
153 
154             // Prorated compared to whole duration
155             availableAmount = (vesting.allocated * (vestedIntervals * vesting.interval)) / vesting.
                    duration;
156 
157             availableAmount -= vesting.released;
158         }
159 
160         uint256 tokenBalance = token.balanceOf(address(this));
161 
162         // just in case if rounding error causes pool to not have enough token.
163         if (availableAmount > tokenBalance) {
164             availableAmount = tokenBalance;
165         }
166 
167         return availableAmount;
168     }
```

Listing 2.11: MintedVesting.sol

Impact N/A

Suggestion Revise the code.

2.3.4 Avoid duplicate calculation

Status Confirmed

Introduced by `Version 1`

Description In the `revoke()` function, the `calculateVestingAmount()` function and the `_withdraw()` function are invoked in line 179 and line 180, respectively.

```
174     function revoke(address _receiver) public onlyOwner onlyExisting(_receiver) {
175         Vesting storage vesting = vestingInfo[_receiver];
176 
177         uint256 availableAmount = calculateVestingAmount(_receiver);
178         _withdraw(_receiver, availableAmount);
179 
180         uint256 remainingAmount = vesting.allocated - vesting.released;
181         token.safeTransfer(owner(), remainingAmount);
182 
183         vesting.released = vesting.allocated;
184         vesting.revoked = true;
185 
186         emit TokenRevoked(_receiver, owner(), remainingAmount);
```

```
187     }
```

Listing 2.12: MintedVesting.sol

However, the `calculateVestingAmount()` function is also invoked in the `_withdraw()` function, which is a duplicate calculation.

```
121     function _withdraw(address _receiver, uint256 _amount) private {
122         require(_amount > 0, "amount is zero");
123
124         Vesting storage vesting = vestingInfo[_receiver];
125
126         uint256 availableAmount = calculateVestingAmount(_receiver);
127
128         require(_amount <= availableAmount, "requested amount exceeds available amount");
129
130         token.safeTransfer(_receiver, _amount);
131         vesting.released += _amount;
132
133         emit TokenReleased(_receiver, _amount);
134     }
```

Listing 2.13: MintedVesting.sol

Impact N/A

Suggestion Revise the code.

2.3.5 Be aware of that the `getPendingReward()` function may return inaccurate result

Status Confirmed

Introduced by `Version 1`

Description When `block.number > periodEndBlock`, the returned value could be different if this function is invoked multiple times. That's because the balance of WCRO token in this contract could be different in different calls. However, this does not affect the real WCRO distribution, since `_updateRewards` will save critical state variables into storage.

```
147     function getPendingRewards() external view returns (uint256) {
148         if (mintedBoost == address(0)) {
149             return 0;
150         }
151
152         if (block.number <= periodEndBlock) {
153             (uint256 multiplier, ) = _getMultiplier();
154             return multiplier * currentRewardsPerBlock;
155         } else {
156             // calculate current round reward
157             (uint256 multiplier, ) = _getMultiplier();
158             uint256 pendingRewardsForCurrentRound = multiplier * currentRewardsPerBlock;
159
160             // calculate next round reward
161             uint256 rewards = wcroToken.balanceOf(address(this)) - pendingRewardsForCurrentRound;
162             uint256 tradingRewards = (rewards * mintedBoostShares) / TOTAL_SHARES;
```

```
163             uint256 nextPeriodEndBlock = block.number + rewardDurationInBlocks;
164             uint256 nextRewardPerBlock = (tradingRewards / (nextPeriodEndBlock - periodEndBlock));
165             uint256 pendingRewardsForNextRound = (block.number - periodEndBlock) *
                    nextRewardPerBlock;
166
167             return pendingRewardsForCurrentRound + pendingRewardsForNextRound;
168         }
169     }
```

Listing 2.14: FeeDistributor.sol

Impact N/A

Suggestion N/A