BLOCKSEC

Security Audit
Report for Lista Token

Date: July 1, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	Lista Token

Version History

Version	Date	Description
1.0	July 1, 2024	First release

Signature

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 target of this audit is the code repository of Lista Token[1] of Lista. Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include `lista-token` folder contract only. Specifically, the files covered in this audit include:

```
VeLista.sol
VeListaDistributor.sol
```

Listing 1.1: Audit Scope for this Report

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
Lista Token	Version 1	c78ad1e6c4e89a3e96a0a7728763df99588301d8
	Version 2	01464f3628d991079895c848ba9cded8b35db3ce

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/lista-dao/lista-token/tree/vetoken

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

		Likelihood	
		High	*Low*
Impact	*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

In total, we find **five** potential issues, **one** recommendation and **one** note as follows:

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

ID	Severity	Description	Category	Status
1	High	Incorrect update of AutoLockAmount in function increaseAmount()	Defi Security	Fixed
2	Medium	Potential asset loss due to lack of check on totalSupplyAtWeek	Defi Security	Fixed
3	Low	Lack of check on _startTime	Defi Security	Fixed
4	Low	Lack of minimum value check in function lock()	Defi Security	Fixed
5	Medium	Potential precision loss in function _claimWithToken()	Defi Security	Fixed
6	-	Redundant code	Recommendation	Fixed
7	-	Potential centralization risk	Note	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect update of AutoLockAmount in function increaseAmount()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In the VeLista contract, the function increaseAmount() is used to increase the amount of tokens locked by a user, and accountLockedData records the user's locked token data. At line 226, when the user's autoLock is set to true, the autoLockAmount is updated to _amount. This is incorrect, as _amount represents the amount to be increased, whereas autoLockAmount should represent the total amount after the increase.

```
187   function increaseAmount(uint256 _amount) external {
188       address _account = msg.sender;
189       uint256 weight = balanceOf(_account);
190       require(weight > 0, "no lock data");
191       require(_amount > 0, "invalid amount");
192
193
194       // transfer lista token
195       token.safeTransferFrom(_account, address(this), _amount);
196       // write history total weight
```

```
        _writeTotalWeight();


        AccountData storage _accountData = accountData[_account];
        uint16 currentWeek = getCurrentWeek();
        uint256 oldWeight = balanceOf(_account);


        // update account data
        _accountData.locked += _amount;
        _accountData.lockTimestamp = block.timestamp;


        if (!_accountData.autoLock) {
            uint16 remainWeek = _accountData.lastLockWeek + _accountData.lockWeeks - currentWeek;
            _accountData.lastLockWeek = currentWeek;
            _accountData.lockWeeks = remainWeek;
        }


        uint256 newWeight = _accountData.locked * uint256(_accountData.lockWeeks);


        // update account locked data
        LockedData[] storage lockedDataHistory = accountLockedData[_account];
        LockedData storage lastAccountLockedData = lockedDataHistory[lockedDataHistory.length - 1];
        if (lastAccountLockedData.week == currentWeek) {
            lastAccountLockedData.locked = _accountData.locked;
            lastAccountLockedData.weight = newWeight;
            lastAccountLockedData.autoLockAmount = _accountData.autoLock ? _accountData.locked : 0;
        } else {
            lockedDataHistory.push(LockedData({
                week: currentWeek,
                locked: _accountData.locked,
                weight: newWeight,
                autoLockAmount: _accountData.autoLock ? _amount : 0
            }));
        }


        // update total locked data
        LockedData storage _totalLockedData = totalLockedData[currentWeek];
        _totalLockedData.locked += _amount;
        _totalLockedData.weight += newWeight - oldWeight;
        if (_accountData.autoLock) {
            _totalLockedData.autoLockAmount += _amount;
        } else {
            // update total unlocked data
            totalUnlockedData[currentWeek + _accountData.lockWeeks] += _amount;
        }


        emit LockAmountIncreased(_account, _amount);
```

```
}
```

Listing 2.1: VeLista.sol

Impact The data recorded within the contract is incorrect.

Suggestion Replace `_amount` with `_accountData.locked`.

2.1.2 Potential asset loss due to lack of check on totalSupplyAtWeek

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VeListaDistributor` contract, the function `depositNewReward()` does not ensure that `veLista.totalSupplyAtWeek` of a specific week is not zero when depositing new rewards for this specific week. Therefore, the rewards deposited for this specific week can never be claimed by users since in the function `_claimWithToken()`, the `rewardAmount` will not be added when `veLista.totalSupplyAtWeek(accountWeek)` is zero.

```
function depositNewReward(uint16 _week, TokenAmount[] memory _tokens) external onlyRole(MANAGER) {
    require(_tokens.length > 0, "no tokens0");
    require(_week >= lastDepositWeek, "week must be greater than or equal to last deposit week");
    require(_week < veLista.getCurrentWeek(), "week must be less than current week");
    if (lastDepositWeek == _week) {
        for (uint8 i = 0; i < _tokens.length; ++i) {
            uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
            require(tokenIdx > 0, "token not registered");
            require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
        }
    }


    lastDepositWeek = _week;


    for (uint8 i = 0; i < _tokens.length; ++i) {
        uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
        uint16 tokenWeek = rewardTokens[tokenIdx].startWeek;
        require(tokenIdx > 0, "token not registered");
        require(_week >= tokenWeek, "deposit week must be greater than or equal to token start week");
        require(_tokens[i].amount > 0, "amount must be greater than 0");
        require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");


        weeklyRewards[_week][tokenIdx] = TokenAmount({
            token: _tokens[i].token,
            amount: _tokens[i].amount
        });
        IERC20(_tokens[i].token).safeTransferFrom(msg.sender, address(this), _tokens[i].amount);
    }


    emit DepositReward(_week, _tokens);
}
```

Listing 2.2: VeListaDistributor.sol

```
function _claimWithToken(address _account, address token, uint16 toWeek) private {
    uint16 currentWeek = veLista.getCurrentWeek();
    require(toWeek < currentWeek, "to week must be less than current week");


    uint256 tokenIdx = rewardTokenIndexes[token];
    require(tokenIdx > 0, "token not registered");


    uint16 accountWeek = accountClaimedWeek[_account][token];
    if (accountWeek == 0) {
        accountWeek = rewardTokens[tokenIdx].startWeek;
    }
    require(accountWeek < currentWeek, "no claimable rewards");


    uint256 amount;


    for (; accountWeek <= toWeek; ++accountWeek) {
        TokenAmount memory reward = weeklyRewards[accountWeek][tokenIdx];
        if (reward.amount == 0) {
            continue;
        }
        uint256 accountWeight = veLista.balanceOfAtWeek(_account, accountWeek);
        uint256 totalWeight = veLista.totalSupplyAtWeek(accountWeek);
        if (totalWeight == 0) {
            continue;
        }
        uint256 rewardAmount = reward.amount;


        amount += rewardAmount * accountWeight / totalWeight;
    }


    if (amount > 0) {
        accountClaimedWeek[_account][token] = accountWeek;
        IERC20(token).safeTransfer(_account, amount);
        emit Claimed(_account, token, amount);
    }
}
```

Listing 2.3: VeListaDistributor.sol

Impact The rewards deposited for a specific week can never be claimed by users.

Suggestion Add a check in the function `depositNewReward()` to ensure that `veLista.totalSupplyAtWeek` of a specific week is not zero when depositing new rewards for this specific week.

2.1.3 Lack of check on _startTime

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VeLista` contract, the function `initialize()` does not validate the parameter `_startTime`, which should be greater than or equal to current timestamp. Specifically, the function `getWeek()` calculates the week by the interval between the current timestamp and `startTime`. If an incorrectly small `_startTime` is passed during initialization, the return value of `getWeek()` could be excessively large. Given that the length of the `totalLockedData` array is fixed at 65535, this scenario could potentially lead to an array out-of-bounds error.

```
function initialize(
    address _admin,
    address _manager,
    uint256 _startTime,
    address _token,
    address _penaltyReceiver
) external initializer {
    require(_admin != address(0), "admin is the zero address");
    require(_manager != address(0), "manager is the zero address");
    require(_token != address(0), "lista token is the zero address");
    require(_penaltyReceiver != address(0), "penalty receiver is the zero address");
    __AccessControl_init();


    _setupRole(DEFAULT_ADMIN_ROLE, _admin);
    _setupRole(MANAGER, _manager);
    startTime = _startTime;
    token = IERC20(_token);
    penaltyReceiver = _penaltyReceiver;
}
```

Listing 2.4: VeLista.sol

```
mapping(address => LockedData[]) accountLockedData;
```

Listing 2.5: VeLista.sol

```
function getWeek(uint256 timestamp) public view returns (uint16) {
    uint256 week = (timestamp - startTime) / 1 weeks;
    if (week <= 65535) {
        return uint16(week);
    }
    revert("exceeds MAX_WEEKS");
}
```

Listing 2.6: VeLista.sol

Impact If the `startTime` is set too early, it can potentially lead to an array out-of-bounds error.

Suggestion Add a check to ensure that `startTime` is greater than or equal to the current timestamp.

2.1.4 Lack of minimum value check in function lock()

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VeLista` contract, there is a lack of minimum value check on the `amount` in function `lock()`, which leads to very small `_accountData.locked`. In this case, the penalty can be zero in the function `earlyClaim()` since the calculation of the penalty is `_accountData.locked * uint256(remainWeek) / uint256(MAX_LOCK_WEEKS)` when `autoLock` is false, `_accountData.locked * uint256(_accountData.lockWeeks) / uint256(MAX_LOCK_WEEKS)` when `autoLock` is true.

```solidity
function lock(uint256 amount, uint16 week, bool autoLock) external {
   require(amount > 0, "lock amount must be greater than 0");
   address _account = msg.sender;
   require(accountData[_account].locked == 0, "locked amount must be 0");
   _createLock(_account, amount, week, autoLock);
   token.safeTransferFrom(_account, address(this), amount);
}
```

Listing 2.7: VeLista.sol

```solidity
function earlyClaim() external returns (uint256) {
    address _account = msg.sender;
    uint16 currentWeek = getCurrentWeek();
    AccountData storage _accountData = accountData[_account];
    uint256 weight = balanceOf(_account);
    uint256 locked = _accountData.locked;
    uint16 unlockWeek = _accountData.lastLockWeek + _accountData.lockWeeks;
    bool autoLock = _accountData.autoLock;


    require(_accountData.autoLock || block.timestamp < _accountData.lockTimestamp + uint256(
        _accountData.lockWeeks) * 1 weeks, "cannot claim with penalty");


    uint256 penalty;
    if (!autoLock) {
        uint16 remainWeek = _accountData.lastLockWeek + _accountData.lockWeeks - currentWeek;
        if (remainWeek == 0) {
            remainWeek = 1;
        }
```

```
            penalty = _accountData.locked * uint256(remainWeek) / uint256(MAX_LOCK_WEEKS);
        } else {
            penalty = _accountData.locked * uint256(_accountData.lockWeeks) / uint256(
                MAX_LOCK_WEEKS);
        }
        totalPenalty += penalty;


        uint256 amount = _accountData.locked - penalty;


        // update account data
        _accountData.locked = 0;
        _accountData.autoLock = false;
        _accountData.lastLockWeek = 0;
        _accountData.lockWeeks = 0;
        _accountData.lockTimestamp = 0;


        // update account locked data
        LockedData[] storage lockedDataHistory = accountLockedData[_account];
        LockedData storage lastAccountLockedData = lockedDataHistory[lockedDataHistory.length - 1];
        if (lastAccountLockedData.week == currentWeek) {
            lastAccountLockedData.locked = 0;
            lastAccountLockedData.weight = 0;
            lastAccountLockedData.autoLockAmount = 0;
        } else {
            lockedDataHistory.push(LockedData({
                week: currentWeek,
                locked: 0,
                weight: 0,
                autoLockAmount: 0
            }));
        }
        // update total locked data
        _writeTotalWeight();
        LockedData storage _totalLockedData = totalLockedData[currentWeek];
        if (weight > 0) {
            _totalLockedData.locked -= locked;
            _totalLockedData.weight -= weight;
        }
        if (autoLock) {
            _totalLockedData.autoLockAmount -= locked;
        }


        // update total unlocked data
        totalUnlockedData[currentWeek] += locked;
        if (!autoLock) {
            totalUnlockedData[unlockWeek] -= locked;
        }


        if (amount > 0) {
            token.safeTransfer(_account, amount);
        }


        emit EarlyClaimed(_account, amount, penalty);
        return amount;
    }
```

Listing 2.8: VeLista.sol

Impact The `penalty` could be zero.

Suggestion Add a minimum value check on the `amount` in the function `lock()`.

2.1.5 Potential precision loss in function _claimWithToken()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VeLista` contract, the function `_claimWithToken()` calculates the rewards for users' locked positions and transfers these rewards to the users. At line 259, there is no scaling applied to `rewardAmount`. Specifically, users can claim rewards in multiple tokens, but the decimals of these tokens may vary a lot. In this case, the division operations within the loop amplify precision loss, resulting in users receiving less reward than expected.

```
232    function _claimWithToken(address _account, address token, uint16 toWeek) private {
233        uint16 currentWeek = veLista.getCurrentWeek();
234        require(toWeek < currentWeek, "to week must be less than current week");
235
236
237        uint256 tokenIdx = rewardTokenIndexes[token];
238        require(tokenIdx > 0, "token not registered");
239
240
241        uint16 accountWeek = accountClaimedWeek[_account][token];
242        if (accountWeek == 0) {
243            accountWeek = rewardTokens[tokenIdx].startWeek;
244        }
245        require(accountWeek < currentWeek, "no claimable rewards");
246
247
248        uint256 amount;
249
250
251        for (; accountWeek <= toWeek; ++accountWeek) {
252            TokenAmount memory reward = weeklyRewards[accountWeek][tokenIdx];
253            if (reward.amount == 0) {
254                continue;
255            }
256            uint256 accountWeight = veLista.balanceOfAtWeek(_account, accountWeek);
```

```
        uint256 totalWeight = veLista.totalSupplyAtWeek(accountWeek);
        if (totalWeight == 0) {
            continue;
        }
        uint256 rewardAmount = reward.amount;


        amount += rewardAmount * accountWeight / totalWeight;
    }


    if (amount > 0) {
        accountClaimedWeek[_account][token] = accountWeek;
        IERC20(token).safeTransfer(_account, amount);
        emit Claimed(_account, token, amount);
    }
}
```

Listing 2.9: VeLista.sol

Impact Users may receive less reward than expected.

Suggestion When calculating the `amount`, the `rewardAmount` should be scaled (e.g., `1e18`) to maintain consistency with the precision of `accountWeight` and `totalWeight`. Before invoking the function `safeTransfer()`, scale the `amount` back to the token's corresponding decimal.

2.2 Additional Recommendation

2.2.1 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VeListaDistributor` contract, the conditionals from line 96 to 102 in the function `depositNewReward()` are redundant. Specifically, regardless of whether `lastDepositWeek` equals `_week`, the code will still proceed to the logic at line 106 to 119, which already includes the checks performed from line 96 to 102. The same issue also exists in the function `_writeTotalWeight()`. This function is `private`, and in all places within the current contract where `_writeTotalWeight()` is invoked, its return value is not required. Therefore, having a return value for this function is redundant.

```
92  function depositNewReward(uint16 _week, TokenAmount[] memory _tokens) external onlyRole(
        MANAGER) {
93      require(_tokens.length > 0, "no tokens");
94      require(_week >= lastDepositWeek, "week must be greater than or equal to last deposit week")
            ;
95      require(_week < veLista.getCurrentWeek(), "week must be less than current week");
96      if (lastDepositWeek == _week) {
97          for (uint8 i = 0; i < _tokens.length; ++i) {
98              uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
99              require(tokenIdx > 0, "token not registered");
```

```
                require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
            }
        }


        lastDepositWeek = _week;


        for (uint8 i = 0; i < _tokens.length; ++i) {
            uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
            uint16 tokenWeek = rewardTokens[tokenIdx].startWeek;
            require(tokenIdx > 0, "token not registered");
            require(_week >= tokenWeek, "deposit week must be greater than or equal to token start week");
            require(_tokens[i].amount > 0, "amount must be greater than 0");
            require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");


            weeklyRewards[_week][tokenIdx] = TokenAmount({
                token: _tokens[i].token,
                amount: _tokens[i].amount
            });
            IERC20(_tokens[i].token).safeTransferFrom(msg.sender, address(this), _tokens[i].amount);
        }


        emit DepositReward(_week, _tokens);
    }
```

Listing 2.10: VeListaDistributor.sol

```
    function _writeTotalWeight() private returns (uint256) {
        uint16 currentWeek = getCurrentWeek();


        uint16 updateWeek = lastUpdateTotalWeek;
        if (updateWeek == currentWeek) {
            return totalLockedData[updateWeek].weight;
        }




        LockedData storage lastTotalLockedData = totalLockedData[updateWeek];
        uint256 locked = lastTotalLockedData.locked;
        uint256 weight = lastTotalLockedData.weight;
        uint256 autoLock = lastTotalLockedData.autoLockAmount;
        uint256 decay = locked - autoLock;


        while(updateWeek < currentWeek) {
            ++updateWeek;
            weight -= decay;
            uint256 unlocked = totalUnlockedData[updateWeek];
            if (unlocked > 0) {
                decay -= unlocked;
                locked -= unlocked;
            }
            totalLockedData[updateWeek].weight = weight;
            totalLockedData[updateWeek].autoLockAmount = autoLock;
            totalLockedData[updateWeek].locked = locked;
        }


        lastUpdateTotalWeek = currentWeek;
        return weight;
    }
```

Listing 2.11: VeListaDistributor.sol

Suggestion Remove the redundant code.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the contract, there exists a privileged account that possesses the ability to upgrade the contract via a proxy contract. Additionally, the privileged role known as "`MANAGER`" is authorized to withdraw all ERC20 tokens from the `VeListaDistributor` contract. This functionality is intended for emergency scenarios. However, if the private key associated with the privileged roles is lost or intentionally misused, it carries the risk of potential losses to the protocol.

Feedback from the project We will use multi-sig wallet as `admin`.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS