



Security Audit

Report for Lista Token

Date: July 16, 2024 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction	1
1.1 About Target Contracts	1
1.2 Disclaimer	1
1.3 Procedure of Auditing	2
1.3.1 Software Security	2
1.3.2 DeFi Security	2
1.3.3 NFT Security	3
1.3.4 Additional Recommendation	3
1.4 Security Model	3
Chapter 2 Findings	5
2.1 DeFi Security	5
2.1.1 Incorrect update of <code>AutoLockAmount</code> in function <code>increaseAmount()</code>	5
2.1.2 Potential asset loss due to lack of check on <code>totalSupplyAtWeek</code>	7
2.1.3 Lack of check on <code>_startTime</code>	9
2.1.4 Lack of minimum value check in function <code>lock()</code>	10
2.1.5 Potential precision loss in function <code>_claimWithToken()</code>	12
2.2 Additional Recommendation	13
2.2.1 Redundant code-I	13
2.2.2 Redundant code-II	15
2.2.3 Improper check in function <code>getTokenClaimable()</code>	17
2.3 Note	18
2.3.1 Potential centralization risk	18

Report Manifest

Item	Description
Client	Lista
Target	Lista Token

Version History

Version	Date	Description
1.0	July 1, 2024	First release
2.0	July 16, 2024	Second 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 Token¹ 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:

```
1 VeLista.sol
2 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	77a6fefdc8c03eaad4fee16c48fc824a1b8a657e
	Version 3	3fc3dc36fb38669570357e0b5575624a4683c407

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

Impact	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

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.

²https://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, **three** recommendations and **one** note as follows:

- High Risk: 1
- Medium Risk: 2
- Low Risk: 2
- Recommendation: 3
- Note: 1

ID	Severity	Description	Category	Status
1	High	Incorrect update of <code>AutoLockAmount</code> in function <code>increaseAmount()</code>	Defi Security	Fixed
2	Medium	Potential asset loss due to lack of check on <code>totalSupplyAtWeek</code>	Defi Security	Fixed
3	Low	Lack of check on <code>_startTime</code>	Defi Security	Fixed
4	Low	Lack of minimum value check in function <code>lock()</code>	Defi Security	Fixed
5	Medium	Potential precision loss in function <code>_claimWithToken()</code>	Defi Security	Fixed
6	-	Redundant code-I	Recommendation	Fixed
7	-	Redundant code-II	Recommendation	Fixed
8	-	Improper check in function <code>getTokenClaimable()</code>	Recommendation	Fixed
9	-	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 `VeList` 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
197     _writeTotalWeight();
198
199
200     AccountData storage _accountData = accountData[_account];
201     uint16 currentWeek = getCurrentWeek();
202     uint256 oldWeight = balanceOf(_account);
203
204
205     // update account data
206     _accountData.locked += _amount;
207     _accountData.lockTimestamp = block.timestamp;
208
209
210     if (!_accountData.autoLock) {
211         uint16 remainWeek = _accountData.lastLockWeek + _accountData.lockWeeks - currentWeek;
212         _accountData.lastLockWeek = currentWeek;
213         _accountData.lockWeeks = remainWeek;
214     }
215
216
217     uint256 newWeight = _accountData.locked * uint256(_accountData.lockWeeks);
218
219
220     // update account locked data
221     LockedData[] storage lockedDataHistory = accountLockedData[_account];
222     LockedData storage lastAccountLockedData = lockedDataHistory[lockedDataHistory.length - 1];
223     if (lastAccountLockedData.week == currentWeek) {
224         lastAccountLockedData.locked = _accountData.locked;
225         lastAccountLockedData.weight = newWeight;
226         lastAccountLockedData.autoLockAmount = _accountData.autoLock ? _accountData.locked : 0;
227     } else {
228         lockedDataHistory.push(LockedData({
229             week: currentWeek,
230             locked: _accountData.locked,
231             weight: newWeight,
232             autoLockAmount: _accountData.autoLock ? _amount : 0
233         }));
234     }
235
236
237     // update total locked data
238     LockedData storage _totalLockedData = totalLockedData[currentWeek];
239     _totalLockedData.locked += _amount;
240     _totalLockedData.weight += newWeight - oldWeight;
241     if (_accountData.autoLock) {
242         _totalLockedData.autoLockAmount += _amount;
243     } else {
244         // update total unlocked data
245         totalUnlockedData[currentWeek + _accountData.lockWeeks] += _amount;
```

```
246     }
247
248
249     emit LockAmountIncreased(_account, _amount);
250 }
```

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.

```
92  function depositNewReward(uint16 _week, TokenAmount[] memory _tokens) external onlyRole(
    MANAGER) {
93      require(_tokens.length > 0, "no tokens0");
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");
100             require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
101         }
102     }
103
104
105     lastDepositWeek = _week;
106
107
108     for (uint8 i = 0; i < _tokens.length; ++i) {
109         uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
110         uint16 tokenWeek = rewardTokens[tokenIdx].startWeek;
111         require(tokenIdx > 0, "token not registered");
112         require(_week >= tokenWeek, "deposit week must be greater than or equal to token start
            week");
113         require(_tokens[i].amount > 0, "amount must be greater than 0");
114         require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
115
116     }
```

```
117         weeklyRewards[_week][tokenIdx] = TokenAmount({
118             token: _tokens[i].token,
119             amount: _tokens[i].amount
120         });
121         IERC20(_tokens[i].token).safeTransferFrom(msg.sender, address(this), _tokens[i].amount)
122         ;
123     }
124
125     emit DepositReward(_week, _tokens);
126 }
```

Listing 2.2: VeListaDistributor.sol

```
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);
257         uint256 totalWeight = veLista.totalSupplyAtWeek(accountWeek);
258         if (totalWeight == 0) {
259             continue;
260         }
261         uint256 rewardAmount = reward.amount;
262
263
264         amount += rewardAmount * accountWeight / totalWeight;
265     }
266
267
268     if (amount > 0) {
269         accountClaimedWeek[_account][token] = accountWeek;
270         IERC20(token).safeTransfer(_account, amount);
271     }
```

```
271         emit Claimed(_account, token, amount);
272     }
273 }
```

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.

```
54  function initialize(
55      address _admin,
56      address _manager,
57      uint256 _startTime,
58      address _token,
59      address _penaltyReceiver
60  ) external initializer {
61      require(_admin != address(0), "admin is the zero address");
62      require(_manager != address(0), "manager is the zero address");
63      require(_token != address(0), "lista token is the zero address");
64      require(_penaltyReceiver != address(0), "penalty receiver is the zero address");
65      __AccessControl_init();
66
67
68      _setupRole(DEFAULT_ADMIN_ROLE, _admin);
69      _setupRole(MANAGER, _manager);
70      startTime = _startTime;
71      token = IERC20(_token);
72      penaltyReceiver = _penaltyReceiver;
73  }
```

Listing 2.4: VeLista.sol

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

Listing 2.5: VeLista.sol

```
78 function getWeek(uint256 timestamp) public view returns (uint16) {
79     uint256 week = (timestamp - startTime) / 1 weeks;
80     if (week <= 65535) {
81         return uint16(week);
82     }
83     revert("exceeds MAX_WEEKS");
84 }
```

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.a.locked * uint256(_accountData.lockWeeks) / uint256(MAX_LOCK_WEEKS)` when `autoLock` is true.

```
99 function lock(uint256 amount, uint16 week, bool autoLock) external {
100     require(amount > 0, "lock amount must be greater than 0");
101     address _account = msg.sender;
102     require(accountData[_account].locked == 0, "locked amount must be 0");
103     _createLock(_account, amount, week, autoLock);
104     token.safeTransferFrom(_account, address(this), amount);
105 }
```

Listing 2.7: VeLista.sol

```
504 function earlyClaim() external returns (uint256) {
505     address _account = msg.sender;
506     uint16 currentWeek = getCurrentWeek();
507     AccountData storage _accountData = accountData[_account];
508     uint256 weight = balanceOf(_account);
509     uint256 locked = _accountData.locked;
510     uint16 unlockWeek = _accountData.lastLockWeek + _accountData.lockWeeks;
511     bool autoLock = _accountData.autoLock;
512
513
514     require(_accountData.autoLock || block.timestamp < _accountData.lockTimestamp + uint256(
        _accountData.lockWeeks) * 1 weeks, "cannot claim with penalty");
515 }
```

```
516
517     uint256 penalty;
518     if (!autoLock) {
519         uint16 remainWeek = _accountData.lastLockWeek + _accountData.lockWeeks - currentWeek;
520         if (remainWeek == 0) {
521             remainWeek = 1;
522         }
523         penalty = _accountData.locked * uint256(remainWeek) / uint256(MAX_LOCK_WEEKS);
524     } else {
525         penalty = _accountData.locked * uint256(_accountData.lockWeeks) / uint256(
526             MAX_LOCK_WEEKS);
527     }
528     totalPenalty += penalty;
529
530     uint256 amount = _accountData.locked - penalty;
531
532
533     // update account data
534     _accountData.locked = 0;
535     _accountData.autoLock = false;
536     _accountData.lastLockWeek = 0;
537     _accountData.lockWeeks = 0;
538     _accountData.lockTimestamp = 0;
539
540
541     // update account locked data
542     LockedData[] storage lockedDataHistory = accountLockedData[_account];
543     LockedData storage lastAccountLockedData = lockedDataHistory[lockedDataHistory.length - 1];
544     if (lastAccountLockedData.week == currentWeek) {
545         lastAccountLockedData.locked = 0;
546         lastAccountLockedData.weight = 0;
547         lastAccountLockedData.autoLockAmount = 0;
548     } else {
549         lockedDataHistory.push(LockedData({
550             week: currentWeek,
551             locked: 0,
552             weight: 0,
553             autoLockAmount: 0
554         }));
555     }
556     // update total locked data
557     _writeTotalWeight();
558     LockedData storage _totalLockedData = totalLockedData[currentWeek];
559     if (weight > 0) {
560         _totalLockedData.locked -= locked;
561         _totalLockedData.weight -= weight;
562     }
563     if (autoLock) {
564         _totalLockedData.autoLockAmount -= locked;
565     }
566
567
```

```
568     // update total unlocked data
569     totalUnlockedData[currentWeek] += locked;
570     if (!autoLock) {
571         totalUnlockedData[unlockWeek] -= locked;
572     }
573
574
575     if (amount > 0) {
576         token.safeTransfer(_account, amount);
577     }
578
579
580     emit EarlyClaimed(_account, amount, penalty);
581     return amount;
582 }
```

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 tokenId = rewardTokenIndexes[token];
238     require(tokenId > 0, "token not registered");
239
240
241     uint16 accountWeek = accountClaimedWeek[_account][tokenId];
242     if (accountWeek == 0) {
243         accountWeek = rewardTokens[tokenId].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);
257         uint256 totalWeight = veLista.totalSupplyAtWeek(accountWeek);
258         if (totalWeight == 0) {
259             continue;
260         }
261         uint256 rewardAmount = reward.amount;
262
263
264         amount += rewardAmount * accountWeight / totalWeight;
265     }
266
267
268     if (amount > 0) {
269         accountClaimedWeek[_account][token] = accountWeek;
270         IERC20(token).safeTransfer(_account, amount);
271         emit Claimed(_account, token, amount);
272     }
273 }
```

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-I

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     ;
96     require(_week < veLista.getCurrentWeek(), "week must be less than current week");
97     if (lastDepositWeek == _week) {
98         for (uint8 i = 0; i < _tokens.length; ++i) {
99             uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
100             require(tokenIdx > 0, "token not registered");
101             require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
102         }
103     }
104
105     lastDepositWeek = _week;
106
107     for (uint8 i = 0; i < _tokens.length; ++i) {
108         uint8 tokenIdx = rewardTokenIndexes[_tokens[i].token];
109         uint16 tokenWeek = rewardTokens[tokenIdx].startWeek;
110         require(tokenIdx > 0, "token not registered");
111         require(_week >= tokenWeek, "deposit week must be greater than or equal to token start
112             week");
113         require(_tokens[i].amount > 0, "amount must be greater than 0");
114         require(weeklyRewards[_week][tokenIdx].amount == 0, "reward already deposited");
115
116         weeklyRewards[_week][tokenIdx] = TokenAmount({
117             token: _tokens[i].token,
118             amount: _tokens[i].amount
119         });
120         IERC20(_tokens[i].token).safeTransferFrom(msg.sender, address(this), _tokens[i].amount);
121     }
122
123
124
125     emit DepositReward(_week, _tokens);
126 }
```

Listing 2.10: VeListaDistributor.sol

```
391 function _writeTotalWeight() private returns (uint256) {
392     uint16 currentWeek = getCurrentWeek();
393
394
395     uint16 updateWeek = lastUpdateTotalWeek;
396     if (updateWeek == currentWeek) {
397         return totalLockedData[updateWeek].weight;
398     }
399
400
401
402
403     LockedData storage lastTotalLockedData = totalLockedData[updateWeek];
404     uint256 locked = lastTotalLockedData.locked;
405     uint256 weight = lastTotalLockedData.weight;
```

```
406     uint256 autoLock = lastTotalLockedData.autoLockAmount;
407     uint256 decay = locked - autoLock;
408
409
410     while(updateWeek < currentWeek) {
411         ++updateWeek;
412         weight -= decay;
413         uint256 unlocked = totalUnlockedData[updateWeek];
414         if (unlocked > 0) {
415             decay -= unlocked;
416             locked -= unlocked;
417         }
418         totalLockedData[updateWeek].weight = weight;
419         totalLockedData[updateWeek].autoLockAmount = autoLock;
420         totalLockedData[updateWeek].locked = locked;
421     }
422
423
424     lastUpdateTotalWeek = currentWeek;
425     return weight;
426 }
```

Listing 2.11: VeListaDistributor.sol

Suggestion Remove the redundant code.

2.2.2 Redundant code-II

Status Fixed in [Version 3](#)

Introduced by [Version 2](#)

Description In the [VeLista](#) contract, the check at line 126 is redundant. Specifically, the function `initialize()` has already ensured that `startTime` is less than `block.timestamp`, making this additional check unnecessary.

```
125     function _createLock(address _account, uint256 _amount, uint16 _week, bool autoLock) private {
126         require(block.timestamp >= startTime, "not started");
127         require(_week <= MAX_LOCK_WEEKS, "exceeds MAX_LOCK_WEEKS");
128         require(_week > 0, "invalid lock week");
129
130         // write history total weight
131         _writeTotalWeight();
132         uint16 currentWeek = lastUpdateTotalWeek; // lastUpdateTotalWeek is current week after
            _writeTotalWeight()
133
134         // update account data
135         AccountData storage _accountData = accountData[_account];
136
137         _accountData.locked = _amount;
138         _accountData.lastLockWeek = currentWeek;
139         _accountData.lockWeeks = _week;
140         _accountData.autoLock = autoLock;
141         _accountData.lockTimestamp = block.timestamp;
```

```
142
143     uint256 weight = _amount * uint256(_week);
144
145     // update account locked data
146     LockedData[] storage lockedDataHistory = accountLockedData[_account];
147     if (lockedDataHistory.length == 0) {
148         lockedDataHistory.push(LockedData({
149             week: currentWeek,
150             locked: _amount,
151             weight: weight,
152             autoLockAmount: autoLock ? _amount : 0
153         }));
154     } else {
155         LockedData storage lastAccountLockedData = lockedDataHistory[lockedDataHistory.length -
156             1];
157         if (lastAccountLockedData.week == currentWeek) {
158             lastAccountLockedData.locked = _amount;
159             lastAccountLockedData.weight = weight;
160             lastAccountLockedData.autoLockAmount = autoLock ? _amount : 0;
161         } else {
162             lockedDataHistory.push(LockedData({
163                 week: currentWeek,
164                 locked: _amount,
165                 weight: weight,
166                 autoLockAmount: autoLock ? _amount : 0
167             }));
168         }
169     }
170
171     // update total locked data
172     LockedData storage _totalLockedData = totalLockedData[currentWeek];
173     _totalLockedData.locked += _amount;
174     _totalLockedData.weight += weight;
175     if (autoLock) {
176         _totalLockedData.autoLockAmount += _amount;
177     } else {
178         // update total unlocked data
179         totalUnlockedData[currentWeek + _week] += _amount;
180     }
181
182     emit LockCreated(_account, _amount, _week, autoLock);
183 }
```

Listing 2.12: VeLista.sol

```
54 function initialize(
55     address _admin,
56     address _manager,
57     uint256 _startTime,
58     address _token,
59     address _penaltyReceiver
60 ) external initializer {
61     require(_admin != address(0), "admin is the zero address");
```

```
62     require(_manager != address(0), "manager is the zero address");
63     require(_token != address(0), "lista token is the zero address");
64     require(_penaltyReceiver != address(0), "penalty receiver is the zero address");
65     require(_startTime < block.timestamp && block.timestamp - _startTime <= 10 weeks, "invalid
        start time");
66     __AccessControl_init();
67
68     _setupRole(DEFAULT_ADMIN_ROLE, _admin);
69     _setupRole(MANAGER, _manager);
70     startTime = _startTime;
71     token = IERC20(_token);
72     penaltyReceiver = _penaltyReceiver;
73 }
```

Listing 2.13: VeLista.sol

Suggestion Remove the redundant code.

2.2.3 Improper check in function getTokenClaimable()

Status Fixed in [Version 3](#)

Introduced by [Version 2](#)

Description In the contract [VeListaDistributor](#), the function [getTokenClaimable\(\)](#) is used to calculate the user reward. In line 36, it is suggested to check whether [accountWeight](#) is zero instead of [totalWeight](#). This is because when [accountWeight](#) is zero, the claimable reward for the account that week is also zero, allowing the loop to continue efficiently.

```
171 function getTokenClaimable(address _account, address _token, uint16 toWeek) public view
    returns (uint256, uint16) {
172     uint256 currentWeek = veLista.getCurrentWeek();
173     require(toWeek < currentWeek, "toWeek must be less than the current week");
174     uint256 claimableAmount;
175
176     uint16 accountWeek = accountClaimedWeek[_account][_token];
177     uint16 lastClaimableWeek = accountWeek;
178     uint8 tokenIdx = rewardTokenIndexes[_token];
179     if (tokenIdx == 0) {
180         return (0, 0);
181     }
182     if (accountWeek == 0) {
183         accountWeek = rewardTokens[tokenIdx].startWeek;
184     }
185     for (uint16 j = accountWeek; j <= toWeek; ++j) {
186         TokenAmount memory reward = weeklyRewards[j][tokenIdx];
187         if (reward.amount == 0) {
188             continue;
189         }
190         lastClaimableWeek = j+1;
191         uint256 accountWeight = veLista.balanceOfAtWeek(_account, j);
192         uint256 totalWeight = veLista.totalSupplyAtWeek(j);
193         if (totalWeight == 0) {
194             continue;
```

```
195     }
196     uint256 rewardAmount = reward.amount;
197     claimableAmount += rewardAmount * accountWeight * 1e18 / totalWeight;
198 }
199 return (claimableAmount / 1e18, lastClaimableWeek);
200 }
```

Listing 2.14: VeListaDistributor.sol

Suggestion Revise the condition on line 36 by replacing `totalWeight` with `accountWeight`.

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](#).

