SALUS SECURITY

JUL 2024

CODE SECURITY ASSESSMENT

LISTA DAO

Overview

Project Summary

- Name: Lista Dao - Lista token
- Platform: BNB Smart Chain
- Language: Solidity
- Repository:
  - https://github.com/lista-dao/lista-token
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Lista Dao - Lista token
Version	v4
Type	Solidity
Dates	Jul 12 2024
Logs	Jun 27 2024; Jul 01 2024; Jul 02 2024; Jul 12 2024

Vulnerability Summary

Total High-Severity issues	0
Total Medium-Severity issues	1
Total Low-Severity issues	2
Total informational issues	3
Total	6

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	User may lose reward and the reward tokens will be locked in the contract	Medium	Business Logic	Resolved
2	Spend a small amount of tokens to gain a large amount of voting power	Low	Business Logic	Acknowledged
3	Centralization risk	Low	Centralization	Mitigated
4	Gas optimization suggestions	Informational	Gas Optimization	Resolved
5	Use of floating pragma	Informational	Configuration	Acknowledged
6	Mismatch between comments and code	Informational	Code Quality	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. User may lose reward and the reward tokens will be locked in the contract	
Severity: Medium	Category: Business Logic
Target: - contracts/VeListaDistributor.sol	

Description

When a user calls the _claimWithToken() function to perform a claim operation, the accountClaimedWeek variable in the function will be updated to toWeek + 1.

contracts/VeListaDistributor.sol: L232-L267

```
function _claimWithToken(address _account, address token, uint16 toWeek) private {
    ...
    uint16 accountWeek = accountClaimedWeek[_account][token];
    for (; accountWeek <= toWeek; ++accountWeek) {
        TokenAmount memory reward = weeklyRewards[accountWeek][tokenIdx];
        if (reward.amount == 0) {
            continue;
        }
       ...
    }

    if (amount > 0) {
        accountClaimedWeek[_account][token] = accountWeek;
        ...
    }
}
```

This means that users are not able to claim rewards for the same week repeatedly.

For a given week, if a user claims the reward before the administrator sets the reward, the user will not be able to receive this portion of the reward, and it will be locked in the contract.

Proof of Concept

The following tests show that users may lose rewards.

```
function test_claim() public {
        VeListaDistributor.TokenAmount[] memory tokensAmount = new
VeListaDistributor.TokenAmount[](2);
        tokensAmount[0].token = address(token1);
        tokensAmount[0].amount = 100 ether;
        tokensAmount[1].token = address(token2);
        tokensAmount[1].amount = 1000 ether;
        address[] memory tokens = new address[](2);
        tokens[0] = address(token1);
        tokens[1] = address(token2);

        skip(1 weeks);
        vm.prank(user1);
        veLista.lock(30 ether, 50, true);

        vm.prank(user2);
        veLista.lock(70 ether, 50, true);

        skip(1 weeks);

        vm.startPrank(manager);
        distributor.registerNewToken(address(token1));
        distributor.registerNewToken(address(token2));
        vm.stopPrank();

        skip(10 weeks);
        vm.prank(manager);
        distributor.depositNewReward(veLista.getCurrentWeek() - 5, tokensAmount);

        vm.startPrank(user1);
        distributor.claimAll(tokens, veLista.getCurrentWeek() - 1);
        vm.stopPrank();

        uint256 user1Token1Balance0 = token1.balanceOf(user1);
        uint256 user1Token2Balance0 = token2.balanceOf(user1);

        assertEq(user1Token1Balance0,
            tokensAmount[0].amount * veLista.balanceOfAtWeek(user1,
veLista.getCurrentWeek()-1) / veLista.totalSupplyAtWeek(veLista.getCurrentWeek()-1),
            "user1 has not claimed token1");
        assertEq(user1Token2Balance0,
            tokensAmount[1].amount * veLista.balanceOfAtWeek(user1,
veLista.getCurrentWeek()-1) / veLista.totalSupplyAtWeek(veLista.getCurrentWeek()-1),
            "user1 has not claimed token2");

        vm.prank(manager);
        distributor.depositNewReward(eLista.getCurrentWeek() - 1, tokensAmount);

        vm.prank(user1);
        distributor.claimAll(tokens, veLista.getCurrentWeek() - 1);
        uint256 user1Token1Balance1 = token1.balanceOf(user1);
        uint256 user1Token2Balance1 = token2.balanceOf(user1);

        assertGt(user1Token1Balance1, user1Token1Balance0);
        assertGt(user1Token2Balance1, user1Token2Balance0);
    }
}
```

Recommendation

Consider limiting toWeek <= lastDepositWeek when claiming.

Status

This issue has been resolved by the team with commit 8c716e6.

2. Spend a small amount of tokens to gain a large amount of voting power

Severity: Low	Category: Business Logic
Target: - contracts/VeLista.sol	

Description

The README.md file mentions that users can lock LISTA tokens to obtain governance rights in the LISTA DAO.

```
Given as an incentive for users of the protocol. Can be locked in `TokenLocker`
to receive lock weight, which gives governance power within the Lista DAO.
```

Malicious users can lock a large amount of tokens from flash loan or swap in veLISTA for a week to gain significant voting power. They can then use the earlyClaim function to claim all tokens prematurely, incurring a penalty. This way, malicious users only need to spend 1/52 of the lock amount to obtain 52 times the voting power, thereby influencing governance.

contracts/VeLista.sol: L504-L574

```
function earlyClaim() external returns (uint256) {
    ...
    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;
   ...
}
```

Recommendation

Consider increasing the penalty to `_accountData.locked * uint256(remainWeek) / uint256(_accountData.lockWeeks)` or half to increase the attacker's cost.

Status

This issue has been acknowledged by the team.

3. Centralization risk	
Severity: Low	Category: Centralization
Target: - contracts/VeLista.sol - contracts/VeListaDistributor.sol	

Description

VeLista and VeListaDistributor contract have privileged accounts, privileged accounts can claim penalty, register reward token, deposit reward token and withdraw all reward tokens in the VeListaDistributor contract.

If Manager's private key or admin's is compromised, an attacker can register malicious tokens, grant roles to arbitrary addresses and withdraw reward tokens.

If the privileged accounts are plain EOA accounts, this can be worrisome and pose a risk to the other users.

Recommendation

We recommend transferring privileged accounts to multi-sig accounts with timelock governors for enhanced security. This ensures that no single person has full control over the accounts and that any changes must be authorized by multiple parties.

Status

The team has stated that they will move admin privileges to the multi-sig account 0x8d388136d578dCD791D081c6042284CED6d9B0c6.

2.3 Informational Findings

4. Gas optimization suggestions	
Severity: Informational	Category: Gas Optimization
Target: - contracts/VeListaDistributor.sol - contracts/VeLista.sol	

Description

1. Memory reading saves more gas than storage reading multiple times when the state is not changed. So caching the storage variables in memory and using the memory instead of storage reading is effective. So cache array length outside of the loop can save gas.
contracts/VeListaDistributor.sol:L72

```
for (idx = 1; idx < rewardTokens.length; ++idx) {
```

contracts/VeListaDistributor.sol:L136

```
for (uint8 i = 1; i < rewardTokens.length; ++i) {
```

contracts/VeListaDistributor.sol:L167

```
for (uint8 i = 1; i < rewardTokens.length; ++i) {
```

contracts/VeListaDistributor.sol:L219

```
for (uint256 i = 0; i < tokens.length; ++i) {
```

2.There is no need to cache the same value twice that does not change. This will increase gas use. So using the same cache can save gas.
contracts/VeLista.sol:L187-L200

```
function increaseAmount(uint256 _amount) external {
    address _account = msg.sender;
    uint256 weight = balanceOf(_account);
    ...
    uint256 oldWeight = balanceOf(_account);
}
```

3.The requirement will cause a revert if the `accountData.autoLock` is true. Therefore, the redundant code - ` _accountData.autoLock = false;` should be removed to save gas.
contracts/VeLista.sol:L478-L485

```
function claim() external returns (uint256) {
    ...
    require(!_accountData.autoLock && block.timestamp >= _accountData.lockTimestamp +
uint256(_accountData.lockWeeks) * 1 weeks, "no claimable tokens");
    ...
    _accountData.autoLock = false;
}
```

Recommendation

Consider using the above suggestions to save gas.

Status

This issue has been resolved by the team with commit 8c716e6.

5. Use of floating pragma	
Severity: Informational	Category: Configuration
Target: - contracts/VeListaDistributor.sol - contracts/VeLista.sol	

Description

```
pragma solidity ^0.8.10;
```

The VeListaDistributor and VeLista uses a floating compiler version 0.8.10.

Using a floating pragma 0.8.10 statement is discouraged, as code may compile to different bytecodes with different compiler versions. Use a locked pragma statement to get a deterministic bytecode. Also use the latest Solidity version to get all the compiler features, bug fixes and optimizations.

Recommendation

It is recommended to use a locked Solidity version throughout the project. It is also recommended to use the most stable and up-to-date version.

Status

The team has stated that they will use compiler version 0.8.19 when deploying.

6. Mismatch between comments and code

Severity: Informational	Category: Code Quality
Target: - contracts/VeLista.sol	

Description

The code comment for the VeLista contract is an error. The VeLista contract locks Lista token to get veLista not the current code comment.

src/ORAStakeRouter.sol:L10-L13

```
/**
  * @title VeLista
  * @dev lock veLista token to get veLista (voting power)
  */
```

Recommendation

Rewrite `lock veLista token to get veLista (voting power)` to `locks Lista token to get veLista (voting power)` .

Status

This issue has been resolved by the team with commit c776f20.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit c78ad1e:

File	SHA-1 hash
VeLista.sol	8804118d0b86bb2ad8bd4385c506dc26146d7521
VeListaDistributor.sol	4d9affd24f96715f0caab2dff61e5ae00e75bc31

And we audited the commit 77a6fef that introduced new features lista-dao/lista-token repository:

File	SHA-1 hash
VeLista.sol	55e98b29caa266cca7527f1d080099955ace56b1
VeListaDistributor.sol	6d77b8005b5eff11b443860ca2e234c26fea134e