BLOCKSEC

Security Audit Report for Lista veToken Emission

Date: August 12, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	Lista veToken Emission

Version History

Version	Date	Description
1.0	August 12, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-08-14 12:51:42+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of Lista veToken Emission[1] [2]of Lista. Note that, we did **NOT** audit all the modules in the repository. The modules this audit report covers include `LISTA-DAO-CONTRACTS` and `LISTA-TOKEN` contracts. Specifically, the file covered in this audit includes:

```
LISTA-TOKEN:
library/TickMath.sol
dao/BorrowLisUSDListaDistributor.sol
dao/CommonListaDistributor.sol
dao/ERC20LpListaDistributor.sol
dao/ERC721LpListaDistributor.sol
dao/ListaVault.sol
dao/OracleCenter.sol
dao/SlisBnbDistributor.sol
dao/interfaces/IDistributor.sol
dao/interfaces/INonfungiblePositionManager.sol
dao/interfaces/IVault.sol
dao/interfaces/OracleInterface.sol

LISTA-DAO-CONTRACTS:
contracts/Interaction.sol
contracts/Jar.sol
contracts/interfaces/IStakeLisUSDListaDistributor.sol
contracts/interfaces/IBorrowLisUSDListaDistributor.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.

[1] https://github.com/lista-dao/lista-dao-contracts

[2] https://github.com/lista-dao/lista-token

Repository	Version	Commit Hash
Lista Token	Version 1	5a606743ef1a10de3cfeb5acd23fc2c859676b81
	Version 2	ea31a35deec117492544e424c20aad0865463b55
Lista Dao Contracts	Version 1	a1168299036aedede684fb357d89b114c78df16b

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.

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 [3] and Common Weakness Enumeration [4]. 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.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[4] https://cwe.mitre.org/

Table 1.1: Vulnerability Severity Classification

Impact		Likelihood: High	Likelihood: Low
Impact	*High*	High	Medium
Impact	*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.
- **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 found **four** potential security issues. Besides, we have **two** recommendations and **one** note.

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

ID	Severity	Description	Category	Status
1	High	Lack of access control in function `onERC721Received()`	DeFi Security	Fixed
2	Medium	Incorrect calculation in function `tickToPrice()`	DeFi Security	Fixed
3	Low	Inconsistency between comment and implementation	DeFi Security	Fixed
4	Medium	Incorrect calculation in function `extractDust()`	DeFi Security	Confirmed
5	-	Lack of storage gap in contract `CommonListaDistributor`	Recommendation	Fixed
6	-	Redundant code	Recommendation	Confirmed
7	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of access control in function `onERC721Received()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `ERC721LpListaDistributor` is designed to receive the `ERC721` token as the underlying token to be staked. When the token is transferred in, the callback function `onERC721Received()` will be invoked to update the contract state, such as minting shares for the staker. However, this function does not perform any legitimate checks on the `msg.sender`. In this case, anyone can simply provide a valid `tokenId` to mint unlimited shares and claim rewards for themselves.

```
/**
 * @dev on nft received
 * @param operator operator address
 * @param from from address
 * @param tokenId tokenId of LP token
 * @param data call data
```

```
    */
function onERC721Received(
    address operator,
    address from,
    uint256 tokenId,
    bytes calldata data
) override external returns (bytes4) {
    (bool isValid, uint256 liquidity) = checkNFT(tokenId);
    require(isValid, "invalid NFT");


    _addNFT(from, tokenId, liquidity);
    _deposit(from, liquidity);
    return IERC721Receiver.onERC721Received.selector;
}
```

Listing 2.1: ERC721LpListaDistributor.sol

Impact Rewards of the contract can be drained.

Suggestion Add checks to ensure the `msg.sender` is the `ERC721` token.

2.1.2 Incorrect calculation in function `tickToPrice()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `ERC721LpListaDistributor` contract, the function `tickToPrice()` calculates the corresponding price based on the user's `tickLower` and `tickUpper` of their positions. The function then evaluates whether the user's `position` meets the contract's set price range based on the calculated price.

However, the `tickToPrice()` function currently does not account for the decimal differences in `token0` and `token1`. If the decimal values of these tokens deviate from the standard `1e18`, this oversight could lead to incorrect price calculations, potentially affecting the validity of prices.

```
function tickToPrice(int24 tick) private pure returns (uint256) {
    uint160 sqrtPriceX96 = TickMath.getSqrtRatioAtTick(tick);
    uint256 sqrtPrice = uint256(sqrtPriceX96)* 1e18 / (1 << 96);
    return sqrtPrice * sqrtPrice / 1e18;
}
```

Listing 2.2: ERC721LpListaDistributor.sol

Impact The price corresponding to the tick has not been accurately scaled to `1e18`.

Suggestion Revise the logic to ensure that the price corresponding to the tick is accurately calculated.

2.1.3 Inconsistency between comment and implementation

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The comment for the function setExpireDelay(), which states "Expire delay in seconds", indicates that _expireDelay is measured in seconds. However, in the usage of _expireDelay, it is treated as weeks, which is inconsistent.

```
/**
 * @dev Set the claim expire delay
 * @param _expireDelay Expire delay in seconds
 */
function setExpireDelay(uint256 _expireDelay) external onlyRole(DEFAULT_ADMIN_ROLE) {
    require(_expireDelay != expireDelay, "Already set");
    expireDelay = _expireDelay;


    emit ExpireDelaySet(_expireDelay);
}
```

Listing 2.3: CommonListaDistributor.sol

Impact _expireDelay may be set to an unreasonable value.

Suggestion Ensure the comment and implementation are consistent.

2.1.4 Incorrect calculation in function extractDust()

Severity Medium

Status Confirmed

Introduced by Version 1

Description In the Jar contract, the function extractDust() serves as a privileged function designed to transfer out residual tokens—referred to as "dust"—that are neither part of user assets nor rewards. However, this dust is calculated by comparing the current token balance of the contract with the totalSupply. Specifically, the rewards that users receive after depositing assets through the function join() are also included in the contract's token balance. This calculation overlooks the users' rewards, which is incorrect.

```
function extractDust() external auth {
    require(block.timestamp >= endTime, "Jar/in-distribution");
    uint dust = IERC20Upgradeable(HAY).balanceOf(address(this)) - totalSupply;
    if (dust != 0) {
        IERC20Upgradeable(HAY).safeTransfer(msg.sender, dust);
    }
}
```

Listing 2.4: Jar.sol

Impact Calculating using the token balance and totalSupply overlooks the users' rewards.

Suggestion Revise the logic to ensure that the `dust` calculation does not include the users' rewards.

Feedback from the project The team promise to only invoke the function in emergency situations.

2.2 Additional Recommendation

2.2.1 Lack of storage gap in contract `CommonListaDistributor`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `CommonListaDistributor` is an abstract contract, it lacks a storage gap for feature upgrades.

Suggestion Add storage gap in the contract `CommonListaDistributor`.

2.2.2 Redundant code

Status Confirmed

Introduced by `Version 1`

Description There are several redundant codes or logic in the contract. Specifically, in the function `_updateReward()`, the `uint128()` cast is redundant.

Meanwhile, the function `deposit()` in contract `ERC721LpListaDistributor` is redundant as the user can directly send NFT to the contract and trigger the function `onERC721Received()`.

```
function _updateReward(address _account, uint256 balance, uint256 supply) internal {
    // update reward
    uint256 updated = periodFinish;
    if (updated > block.timestamp) updated = block.timestamp;
    uint256 duration = updated - lastUpdate;
    if (duration > 0) lastUpdate = uint32(updated);
    if (duration > 0 && supply > 0) {
        rewardIntegral += (duration * rewardRate * 1e18) / supply;
    }
    if (_account != address(0)) {
        uint256 integralFor = rewardIntegralFor[_account];
        if (rewardIntegral > integralFor) {
            storedPendingReward[_account] += uint128((balance * (rewardIntegral - integralFor))
                / 1e18);
            rewardIntegralFor[_account] = rewardIntegral;
        }
    }
}
```

Listing 2.5: CommonListaDistributor.sol

```
function deposit(uint256 tokenId) whenNotPaused external {
    require(IERC721(lpToken).ownerOf(tokenId) == msg.sender, "Not owner of token");
    IERC721(lpToken).safeTransferFrom(msg.sender, address(this), tokenId);
}
```

Listing 2.6: CommonListaDistributor.sol

Suggestion Remove the redundant code.

Feedback from the project Users will use `deposit()` in our website.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the `ListaVault` contract, the `owner` can withdraw the assets within the protocol via function `emergencyWithdraw()`. If the `owner` account's private key is lost or maliciously exploited, it could cause significant damage to the protocol.

Feedback from the project The `owner` will be a multi-sig.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS