BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Lista DAO

USDT Distributor

December 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Lista DAO - USDT Distributor
Website	Lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/lista-token/blob/fd80e186716de76ff22e4dcec89768e4ebecffe2/contracts/dao/USDTLpListaDistributor.sol https://github.com/lista-dao/lista-token/blob/fd80e186716de76ff22e4dcec89768e4ebecffe2/contracts/dao/CommonListaDistributor.sol
Resolution 1	

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	2			
Medium				
Low	5			
Informational	9			
Governance				
Total	16			

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

CommonListaDistributor

The CommonListaDistributor contract distributes Lista rewards from the ListaVault to users in the UDST Distributor mechanism. It implements a staking vault similar to the Synthetix system, managing user deposits, withdrawals, and rewards.

The vault handles only Lista tokens. Weekly, the ListaVault's allocateNewEmissions function specifies the Lista tokens available for distribution, which are gradually released over one week. While ListaVault handles token transfers, the distributor contract tracks pending rewards, updates reward debts during deposits/withdrawals, and informs the ListaVault of the exact reward amounts.

Users can claim rewards directly via this contract or through the ListaVault, which calls this contract to disburse tokens.

Appendix: Reward mechanism.

The reward per share is tracked in the rewardIntegral variable. This is a monotonously increasing quantity which goes up by the amount of dispensed rewards per unit supply.

*rewardIntegral += (duration * rewardRate * 1e18) / supply;*

The system records the rewardIntegral of each user during their deposits or withdrawals. The amount of rewards to be given out is calculated from the difference between the global and user-specific stored reward integrals.

*(balance * (rewardIntegral - rewardIntegralFor[account])) / 1e18;*

Appendix: Core Invariants

INV 1: The rewardIntegral must increase monotonously

INV2: Users cannot claim rewards for the same duration multiple times

INV3: Total claimable incentives must not exceed incentives added

INV4: Incentives must be distributed to users based on their balances

INV5: Rewards for a week can only be fetched once

Privileged Functions

- vaultClaimReward
- notifyRegisteredId
- togglePause
- pause

Issue_01	claimReward doesn't implement whenNotPaused check
Severity	**Low**
Description	The contracts handle two rewards: CAKE and LISTA. The Cake handling vault implements a whenNotPaused check in the claimStakeReward function, preventing the reward collection on pause. However the same is not present in the claimReward function handling Lista rewards. Thus pausing the contract will prevent CAKE rewards from being distributed, but will not pause the LISTA rewards.
Recommendations	Consider adding the whenNotPaused check in the claimReward function.
Comments / Resolution	

Issue_02	whenNotPaused modifier is missing on fetchRewards
Severity	**Low**
Description	*function fetchRewards() external* The fetchRewards function is missing the whenNotPaused modifier, allowing it to be called during a paused state. The function includes a call to updateReward which is blocked by other functions except fetchRewards in the paused state. Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider adding the whenNotPaused modifier to the fetchRewards function.
Comments / Resolution	

USDTLpListaDistributor

The USDTLpListaDistributor contract extends CommonListaDistributor, handling token transfers for deposits and withdrawals. It also manages CAKE token rewards earned by providing and staking LP in PancakeSwap pools. Using a Synthetix-like mechanism, it tracks reward debts and pending CAKE rewards, enabling independent tracking and disbursement of both Lista and CAKE tokens.

This contract facilitates depositing and staking in PancakeSwap pools. CAKE rewards are harvested hourly and distributed over a week. While USDT deposits and withdrawals are managed here, CAKE tokens are stored and distributed by the external StakingVault, based on data from the distributor contract.

Users can claim CAKE rewards directly from this contract or via the StakingVault, which interacts with this contract for disbursement.

Appendix: External Vaults

Two external vaults are used for handling reward tokens: The ListaVault and the StakingVault. The ListaVault contains the Lista tokens, and informs the distributor contract the allocated rewards for the coming week. When rewards are to be paid out, the distributor calls the ListaVault with the information, and the Lista tokens are then moved from the ListaVault to the user.

Similar to the ListaVault, the StakingVault handles the CAKE tokens. Accrued CAKE rewards are sent there and the StakingContract updates the distributor contract with the allocated rewards and keeps a fee. When the rewards are to be paid out, similar to the ListaVault, the distributor calls the StakingVault which transfers the tokens to the user.

The contracts can be found here:

Staking Vault:
https://github.com/lista-dao/lista-token/blob/fd80e186716de76ff22e4dcec89768e4ebecffe2/contracts/dao/StakingVault.sol

ListaVault:
https://github.com/lista-dao/lista-token/blob/e570e131a886a617f15522ed05e0526cfc5acd35/contracts/dao/ListaVault.sol

Appendix: PancakeSwap

The liquidity is provided to PancakeSwap stableswap pool. This pool accepts single sided liquidity, so it is possible to provide liquidity with only USDT tokens. The LP tokens received are then staked in the V2Wrapper contract, which drips CAKE rewards. A PancakeStableSwapTwoPoolInfo contract is also used to pre-calculate the token amounts before deposit/withdrawal.

The contracts can be found here on the BNB chain:

StableswapPool: 0xb1da7d2c257c5700612bde35c8d7187dc80d79f1

StableSwapPoolInfo: 0x150c8AbEB487137acCC541925408e73b92F39A50

V2Wrapper: 0xd069a9E50E4ad04592cb00826d312D9f879eBb02

Stableswap LPToken: 0xB2Aa63f363196caba3154D4187949283F085a488

Appendix: Emergency mode

The contract implements an emergency mode. This is to be enabled if the underlying PancakeSwap mechanism breaks for some reason and leverages the emergencyWithdraw mechanism of the PancakeSwap vault. This forfeits all pending rewards and withdraws the underlying tokens from the pancake vault. This mode also bypasses all reward calculations done in the contract.

Appendix: Core Invariants

INV 1-4: Same invariants 1-4 from CommonListaDistributor

INV 5: Harvest calls need to be at least 1 hour apart

INv 6: Deposits need to always be withdrawable even in emergency mode

Privileged Functions

- notifyStakingReward
- vaultClaimStakingReward
- setStakeVault
- stopEmergencyMode
- emergencyWithdraw
- setHarvestTimeGap
- setIsActive

Issue_03	_updateStakeReward uses incorrect balance for updates
Severity	**High**
Description	The lista distributor calls _updateReward on the balance and totalSupply before updating them during the deposit/withdrawal. However, the _updateStakeReward call is done in the _stakeLp function, which is after the balance updates are done by the _deposit function. So when the stake reward is being updated, the contract uses the balance and supply after the update, not the one before the deposit/withdraw. *_withdraw(msg.sender, lpAmount);* *_unstakeLp(msg.sender, lpAmount);* This leads to issues since according to the staking logic the initial values need to be used. The impact can be shown via a complete withdrawal, where the balance of the user drops to zero. In this case, the _withdraw function updates the Lista reward state based on the old balance and totalSupply, and then the balance changes to 0. Now when _unstakeLp is called, the balance of the account is already 0, and so the _updateStakeReward function calculates the stakeStoredPendingReward value with the current balance (0) and is thus 0. Thus all the staking rewards accrued were lost and set as 0.
Recommendations	_updateStakeReward should be using the pre-update balances and supplies, like in the Lista staking contract. Log the initial balance and totalSupply, and use that to update the staking state instead of the current balance after the storage update.
Comments / Resolution	

Issue_04	emergencyMode breaks reward accounting
Severity	**High**
Description	When in emergency mode, _unstakeLp is not called. So if a user withdraws some tokens during the emergency mode, their balance will decrease but their reward debt will not be updated. So if a user had 100 tokens and withdrew 50 during emergency, and then deposited 50 again after the emergency mode was over, they would have accrued yield on their initial balance of 100 even though they had reduced their stake sometime in between. This way the user can get more rewards than they were owed. The issue is primarily because the stopEmergencyMode function does not update the reward debts and the contract keeps operating with the older reward debts logged before the start of the emergency mode. Any deposits/withdrawals in the emergency period thus break the accounting. Therefore the current version of the contract does not support coming out of the emergency mode and resuming normal operations, even though the contract implements a stopEmergencyMode function.
Recommendations	Going into emergency mode has bad consequences on staking accounting. Before normal operations, the reward debts of all the users need to be set correctly, which is not done in the current stopEmergencyMode implementation. Re-deployment of the distributor contract may be necessary in order to reset the reward debt of all users.
Comments / Resolution	

Issue_05	Short-duration updates could cause precision loss
Severity	**Low**
Description	Reward integral updates involve multiplying the duration by the reward rate and dividing it by the current total supply. Even though the numerator is normalized to 1e18 decimals, there is possible precision loss if the duration is too short, depending on the total supply and the reward rate. For the given example: 1. We have a period of 1 week - 604_800 seconds and 100_000e18 rewards, giving us a reward rate of 165343915343915330 and a staker before the reward notification with 10_000 deposit 2. At the half point, 302_400, we deposit 10_000 more tokens, making our reward integral 302_400 * 165343915343915330 / 10_000 = 5000000000000000000 3. At an uneven timestamp before the end of the period, eg 302_327 seconds since step 1, the first depositor decides to claim and gets a new interval of 302_327 * 165343915343915330 / 20_000 = 2499396494708994600 The period finishes, the first depositor claims again and gets an interval of 73 * 165343915343915330 / 20_000 = 603505291005290.9, which due to precision loss in solidity will become 603505291005290, leaving some rewards unclaimed when the reward calculation integral * balance occurs
Recommendations	Consider keeping this issue in mind and having a governance-controlled recovery function in the vault holding CAKE tokens so any dust can be recovered.
Comments / Resolution	

Issue_06	Rewards accrued before the first deposit will be left unclaimed
Severity	**Low**
Description	The reward mechanism present in both Lista and Cake reward accrual mechanisms are inspired by the Synthethix reward mechanism, which contains a flaw regarding reward calculation if no stakes happen in the same block that the rewards are notified - https://0xmacro.com/blog/synthetix-staking-rewards-issue-inefficient-reward-distribution/ In the current context of the codebase, the Lista/CAKE rewards can get fetched/notified before the first stake happens, which can lead to the following sequence: 1. Rewards get fetched before any stakes occur at timestamp X. 2. At timestamp Y somebody stakes for the first time. Due to the supply > 0 clause for updating the supply, we cannot increase the reward integral. However the lastUpdate variable gets set to Y Any consequent stakes after Y will increase the integral, however the rewards for the X-Y period (between X and Y) will never get distributed to the stakers
Recommendations	Consider keeping this issue in mind and having a recovery function in the StakingVault contract to recover any unused CAKE tokens.
Comments / Resolution	

Issue_07	CAKE rewards can remain inaccessible for the first depositor under certain conditions
Severity	**Low**
Description	stakeReward is only updated in notifyStakingReward i.e. when cake rewards are earned. For the first depositor, there will be no CAKE rewards being dripped to them, since no rewards have been collected yet. In this situation, if the first depositor decides to exit their position, with no other deposit, withdraw, or harvest function calls in between, they will receive no rewards no matter how long they stay in the vault. This is because notifyStakingReward will never get called, so the stakeRewardRate will stay at 0. So the first user can enter a vault, spend a week and then exit a vault with no CAKE rewards, if no other function calls take place in between.
Recommendations	The rewards are always dripped in the future, so this is inherent to the design of the system. To mitigate this loss to an extent, the governance should consider setting up a bot to regularly call harvest on the vault. This will keep the rewards up-to-date and make sure the users get the maximum of rewards possible even if there is low activity in the vaults.
Comments / Resolution	

Issue_08	noHarvest timestamp inequality is not consistent with other timing-based functions
Severity	**Informational**
Description	The noHarvest function will return true if *lastHarvestTime + harvestTimeGap > block.timestamp;* However other timestamp-based functions in the codebase use >= when evaluating the boolean value, such as fetchRewards. *require(getWeek(block.timestamp) >= getWeek(periodFinish),* As we can see here we are using >= to determine the boolean value.
Recommendations	Consider changing > to >= in noHarvest.
Comments / Resolution	

Issue_09	vault address cannot be reset after initialization
Severity	**Informational**
Description	The Lista reward vault address is set in the initialize function. There is no way in the contract to change or reset this value. In contrast, the stakeVault value is also set in the initialize function but can be reset later with the setStakeVault function.
Recommendations	Consider adding a setter for the vault address as well.
Comments / Resolution	

Issue_10	No zero check on _pauser address
Severity	**Informational**
Description	Missing 0 address check on the _pauser address in the initialize function can result in an erroneous pauser being set.
Recommendations	Consider implementing a check for the 0 address when setting the pauser.
Comments / Resolution	

Issue_11	Inconsistent use of access controls
Severity	**Informational**
Description	The CommonListaDistributor contract uses role-based access control for reward claims from vaults, while the USDTLpListaDistributor contract uses direct address-based checks.
Recommendations	Consider using similar setups for access control for both types of reward claims from vaults. Alternatively, this issue can be acknowledged.
Comments / Resolution	

Issue_12	Inconsistencies in slippage checks
Severity	**Informational**
Description	The slippage checks in withdraw can be seen below *require(minLisUSDAmount <= expectLisUSDAmount, "Invalid lisUSD amount");* *require(minUSDTAmount <= expectUSDTAmount, "Invalid USDT amount");* However, in the same function, the second slippage check is as follows *require(lisUSDAmountActual >= minLisUSDAmount, "Invalid lisUSD amount received");* *require(usdtAmountActual >= minUSDTAmount, "Invalid USDT amount received");* While the check is essentially the same in its function, it may mess with the code uniformity.
Recommendations	Consider changing the slippage checks to match the same inequality

	to make the code more uniform.
Comments / Resolution	

Issue_13	setStakeVault does not check rewardToken == v2 wrapper rewardToken
Severity	**Informational**
Description	in the initialize function there is the following check *v2wrapper.rewardToken() == IStakingVault(stakeVault).rewardToken()* However when setting a new stakeVault in setStakeVault, no such check exists, it simply checks if the stakeVault address is not 0.
Recommendations	Consider adding the same check that is included in the initialize function to the setStakeVault function as well.
Comments / Resolution	

Issue_14	setHarvestTimeGap does not check if the value to be set is already set
Severity	**Informational**
Description	setHarvestTimeGap is a setter function which allows the manager to set the value of the harvest time gap. However currently the function allows the manager to set a value which may already be set as the harvest time gap.
Recommendations	If the value being set is already the value that is presently set as the harvest time gap, revert the call.
Comments / Resolution	

Issue_15	updated is unnecessarily cast to uint32
Severity	**Informational**
Description	in the _updateStakeReward, the updated variable is unnecessarily casted to uint32.
Recommendations	Consider not casting the updated variable to uint32 as it is not needed.
Comments / Resolution	

Issue_16	Redundant check in deposit function
Severity	**Informational**
Description	The code checks for slippage thrice. Once with the poolInfo contract, *uint256 expectLpAmount = getLpToMint(usdtAmount);* *require(expectLpAmount >= minLpAmount, "l");* again with the actual stableswap deposit, which also takes the slippage parameter, *IStableSwap(stableSwapPool).add_liquidity([0, usdtAmount], minLpAmount);* and another time in the end. *require(actualLpAmount >= minLpAmount);* Checking the same thing 3 times is unnecessary and leads to excessive gas use.
Recommendations	Consider removing the first check (with expectLpAmount). The last check can be left in the code as a final sanity check.
Comments / Resolution	