BLOCKSEC

Security Audit

Report for USDTLpLista-Distributor

Date: Dec 18, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	USDTLpListaDistributor

Version History

Version	Date	Description
1.0	Dec 18, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-12-18 04:07:21+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 USDTLpListaDistributor[1] of Lista. Specifically, only the changed part in this pull request [2] is in the audit scope.

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
USDTLpListaDistributor	`Version 1`	`fd80e186716de76ff22e4dcec89768e4ebecffe2`
	`Version 2`	`2fd2cfc261a65931a5400c8df978d24d646179a1`

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

[2] https://github.com/lista-dao/lista-token/pull/65

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.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
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.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

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

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **three** potential issues. Besides, we also have **two** recommendations and **three** note.

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

ID	Severity	Description	Category	Status
1	High	Incorrect stake reward calculation due to use of updated balance	DeFi Security	Fixed
2	Medium	Potential lose rewards for users due to incorrect update during emergency mode switch	DeFi Security	Fixed
3	Medium	Potential drain of stake rewards due to incorrect update during emergency mode switch	DeFi Security	Fixed
4	-	Lack of check in function `_setIsActive()`	Recommendation	Fixed
5	-	Lack of reentrancy guard in the `deposit()` and `withdraw()` functions	Recommendation	Fixed
6	-	potential centralization risk	Note	-
7	-	Reward distribution mechanism and potential delay	Note	-
8	-	Expected reward distribution behavior during emergency mode	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect stake reward calculation due to use of updated balance

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `USDTLpListaDistributor` contract, rewards are distributed based on the actual staked `LP` tokens and their duration. When a user makes a deposit, the contract first executes `_deposit()` function to update the user's balance, and then during the `_stakeLp()` function call, it triggers the function `_updateStakeReward()` to update the user's reward. However, in function `_updateStakeReward()`, the updated balance is used for calculating the stake reward, while the newly added `LP` tokens have not yet been staked in the farming pool to generate rewards. This causes the reward calculation to be inflated during the deposit process. Similarly, during withdrawal, the reward calculation will be underestimated for the same reason.

```
270 function _stakeLp(address _account, uint256 _amount) private {
271   address distributor = address(this);
272
273   _updateStakeReward(_account, balanceOf[_account]);
274
275   uint256 beforeBalance = IERC20(cake).balanceOf(distributor);
276   bool _noHarvest = noHarvest();
277   if (!_noHarvest) {
278     lastHarvestTime = block.timestamp;
279   }
280   // Stake LP tokens to V2Wrapper
281   IERC20(lpToken).safeIncreaseAllowance(address(v2wrapper), _amount);
282   v2wrapper.deposit(_amount, _noHarvest);
283
284   uint256 claimed = IERC20(cake).balanceOf(distributor) - beforeBalance;
285
286   // Send CAKE rewards to StakingVault
287   if (claimed > 0) {
288     IERC20(cake).safeIncreaseAllowance(stakeVault, claimed);
289     IStakingVault(stakeVault).sendRewards(distributor, claimed);
290     emit Harvest(address(usdt), distributor, claimed);
291   }
292 }
```

Listing 2.1: USDTLpListaDistributor.sol

```
295 function _unstakeLp(address _account, uint256 amount) private {
296   _updateStakeReward(_account, balanceOf[_account]);
297
298   address distributor = address(this);
299
300   // withdraw lp token and claim rewards
301   uint256 beforeBalance = IERC20(cake).balanceOf(distributor);
302
303   bool _noHarvest = noHarvest();
304   if (!_noHarvest) {
305     lastHarvestTime = block.timestamp;
306   }
307   v2wrapper.withdraw(amount, _noHarvest);
308
309   uint256 claimed = IERC20(cake).balanceOf(distributor) - beforeBalance;
310
311   // Send CAKE rewards to StakingVault
312   if (claimed > 0) {
313     IERC20(cake).safeIncreaseAllowance(stakeVault, claimed);
314     IStakingVault(stakeVault).sendRewards(distributor, claimed);
315     emit Harvest(address(usdt), distributor, claimed);
316   }
317 }
```

Listing 2.2: USDTLpListaDistributor.sol

Impact This inconsistency can lead to inaccurate reward distribution.

Suggestion Revise the logic to pass the correct balance parameter when updating stake rewards.

2.1.2 Potential lose rewards for users due to incorrect update during emergency mode switch

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `USDTLpListaDistributor` contract, there is a security issue related to potential reward loss for users during `emergency mode`. When a user deposits funds via the `deposit()` function, the received `LP` tokens are staked to earn `CAKE` rewards. If the system enters emergency mode and the user later withdraws all staked `LP` tokens via the `withdraw()` function, the absence of a call to `_unstakeLp()` results in the user's stake rewards not being updated. Consequently, the user loses all rewards accrued between the last stake update and the withdrawal.

```
195 function withdraw(uint256 lpAmount, uint256 minLisUSDAmount, uint256 minUSDTAmount) external {
196   require(lpAmount > 0, "Invalid LP amount");
197
198   // 1. Validate lisUSD and USDT amount
199   (uint256 expectLisUSDAmount, uint256 expectUSDTAmount) = getCoinsAmount(lpAmount);
200   require(minLisUSDAmount <= expectLisUSDAmount, "Invalid lisUSD amount");
201   require(minUSDTAmount <= expectUSDTAmount, "Invalid USDT amount");
202
203   // 2. Update user's LP balance and LISTA reward, and distributor's LP total supply
204   _withdraw(msg.sender, lpAmount);
205
206   // 3. Unstake LP token from farming contract if not in emergency mode
207   if (!emergencyMode) _unstakeLp(msg.sender, lpAmount);
208
209   // 4. Remove liquidity from PancakeStableSwapTwoPool
210   uint256 lisUSDAmountActual = lisUSD.balanceOf(address(this));
211   uint256 usdtAmountActual = usdt.balanceOf(address(this));
212
213   IStableSwap(stableSwapPool).remove_liquidity(lpAmount, [minLisUSDAmount, minUSDTAmount]);
214
215   lisUSDAmountActual = lisUSD.balanceOf(address(this)) - lisUSDAmountActual;
216   usdtAmountActual = usdt.balanceOf(address(this)) - usdtAmountActual;
217   require(lisUSDAmountActual >= minLisUSDAmount, "Invalid lisUSD amount received");
218   require(usdtAmountActual >= minUSDTAmount, "Invalid USDT amount received");
219
220   // 5. Transfer lisUSD and USDT to user
221   lisUSD.safeTransfer(msg.sender, lisUSDAmountActual);
222   usdt.safeTransfer(msg.sender, usdtAmountActual);
223
224   emit LpUnstaked(lpToken, usdtAmountActual, lisUSDAmountActual, lpAmount);
225 }
```

Listing 2.3: USDTLpListaDistributor.sol

```
295 function _unstakeLp(address _account, uint256 amount) private {
296   _updateStakeReward(_account, balanceOf[_account]);
297
298   address distributor = address(this);
299
300   // withdraw lp token and claim rewards
301   uint256 beforeBalance = IERC20(cake).balanceOf(distributor);
302
303   bool _noHarvest = noHarvest();
304   if (!_noHarvest) {
305     lastHarvestTime = block.timestamp;
306   }
307   v2wrapper.withdraw(amount, _noHarvest);
308
309   uint256 claimed = IERC20(cake).balanceOf(distributor) - beforeBalance;
310
311   // Send CAKE rewards to StakingVault
312   if (claimed > 0) {
313     IERC20(cake).safeIncreaseAllowance(stakeVault, claimed);
314     IStakingVault(stakeVault).sendRewards(distributor, claimed);
315     emit Harvest(address(usdt), distributor, claimed);
316   }
317 }
```

Listing 2.4: USDTLpListaDistributor.sol

```
330 function _updateStakeReward(address _account, uint256 _balance) internal {
331   // update reward
332   uint256 updated = stakePeriodFinish;
333   if (updated > block.timestamp) updated = block.timestamp;
334   uint256 duration = updated - stakeLastUpdate;
335   if (duration > 0) stakeLastUpdate = uint32(updated);
336
337   if (duration > 0 && totalSupply > 0) {
338     stakeRewardIntegral += (duration * stakeRewardRate * 1e18) / totalSupply;
339   }
340   if (_account != address(0)) {
341     uint256 integralFor = stakeRewardIntegralFor[_account];
342     if (stakeRewardIntegral > integralFor) {
343       stakeStoredPendingReward[_account] += (_balance * (stakeRewardIntegral - integralFor)) / 1
          e18;
344       stakeRewardIntegralFor[_account] = stakeRewardIntegral;
345     }
346   }
347 }
```

Listing 2.5: USDTLpListaDistributor.sol

Impact During `emergency mode`, users may lose rewards accrued during the period between the last stake update and the withdrawal.

Suggestion Modify the `withdraw()` function logic to ensure that stake rewards are appropriately updated in `emergency mode`.

2.1.3 Potential drain of stake rewards due to incorrect update during emergency mode switch

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `USDTLpListaDistributor` contract, when users perform deposit operations during the `emergency mode`, the `_stakeLp()` function is not triggered. This results in the user's stake rewards not being updated promptly. In this case, when the system exits `emergency mode`, subsequent withdrawal operations will cause the `_updateStakeReward()` function to use the user's total current balance (including the amount deposited during `emergency mode`) to calculate rewards for the entire period, which is incorrect.

```
164 function deposit(uint256 usdtAmount, uint256 minLpAmount) external onlyActive {
165   require(usdtAmount > 0, "Invalid usdt amount");
166   uint256 expectLpAmount = getLpToMint(usdtAmount);
167   require(expectLpAmount >= minLpAmount, "Invalid min lp amount");
168
169   // 1. Transfer USDT to this contract
170   usdt.safeIncreaseAllowance(stableSwapPool, usdtAmount);
171   usdt.safeTransferFrom(msg.sender, address(this), usdtAmount);
172
173   // 2. Add USDT to PancakeStableSwapTwoPool
174   uint256 actualLpAmount = IERC20(lpToken).balanceOf(address(this));
175   IStableSwap(stableSwapPool).add_liquidity([0, usdtAmount], minLpAmount);
176   actualLpAmount = IERC20(lpToken).balanceOf(address(this)) - actualLpAmount;
177
178   require(actualLpAmount >= minLpAmount, "Invalid lp amount minted");
179
180   // 3. Update user's LP balance and LISTA reward, and distributor's LP total supply
181   _deposit(msg.sender, actualLpAmount);
182
183   // 4. Stake the received LP token to farming contract only if not in emergency mode
184   if (!emergencyMode) _stakeLp(msg.sender, actualLpAmount);
185
186   emit USDTStaked(lpToken, usdtAmount, actualLpAmount);
187 }
```

Listing 2.6: USDTLpListaDistributor.sol

```
195 function withdraw(uint256 lpAmount, uint256 minLisUSDAmount, uint256 minUSDTAmount) external {
196   require(lpAmount > 0, "Invalid LP amount");
197
198   // 1. Validate lisUSD and USDT amount
199   (uint256 expectLisUSDAmount, uint256 expectUSDTAmount) = getCoinsAmount(lpAmount);
200   require(minLisUSDAmount <= expectLisUSDAmount, "Invalid lisUSD amount");
201   require(minUSDTAmount <= expectUSDTAmount, "Invalid USDT amount");
202
203   // 2. Update user's LP balance and LISTA reward, and distributor's LP total supply
204   _withdraw(msg.sender, lpAmount);
205
```

```
206   // 3. Unstake LP token from farming contract if not in emergency mode
207   if (!emergencyMode) _unstakeLp(msg.sender, lpAmount);
208
209   // 4. Remove liquidity from PancakeStableSwapTwoPool
210   uint256 lisUSDAmountActual = lisUSD.balanceOf(address(this));
211   uint256 usdtAmountActual = usdt.balanceOf(address(this));
212
213   IStableSwap(stableSwapPool).remove_liquidity(lpAmount, [minLisUSDAmount, minUSDTAmount]);
214
215   lisUSDAmountActual = lisUSD.balanceOf(address(this)) - lisUSDAmountActual;
216   usdtAmountActual = usdt.balanceOf(address(this)) - usdtAmountActual;
217   require(lisUSDAmountActual >= minLisUSDAmount, "Invalid lisUSD amount received");
218   require(usdtAmountActual >= minUSDTAmount, "Invalid USDT amount received");
219
220   // 5. Transfer lisUSD and USDT to user
221   lisUSD.safeTransfer(msg.sender, lisUSDAmountActual);
222   usdt.safeTransfer(msg.sender, usdtAmountActual);
223
224   emit LpUnstaked(lpToken, usdtAmountActual, lisUSDAmountActual, lpAmount);
225 }
```

Listing 2.7: USDTLpListaDistributor.sol

Impact This leads to funds deposited during `emergency mode` receiving rewards for periods prior to their `deposit`, resulting in over-allocation of rewards.

Suggestion Revise the logic to ensure correct rewards distribution.

2.2 Additional Recommendation

2.2.1 Lack of check in function `_setIsActive()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `setIsActive()` allows the privileged `MANAGER` role to change the state of the global variable `isActive`. However, it does not check whether the updated state is different from the original state. If the state remains unchanged, the emitted event after the update becomes meaningless.

```
422 function setIsActive(bool _isActive) external onlyRole(MANAGER) {
423   isActive = _isActive;
424   emit SetIsActive(_isActive);
425 }
```

Listing 2.8: USDTLpListaDistributor.sol

Suggestion Add a check to ensure the updated state is different from the original state.

2.2.2 Lack of reentrancy guard in the `deposit()` and `withdraw()` functions

Status Fixed in `Version 2`

Introduced by Version 1

Description In the USDTLpListaDistributor contract, the deposit() and withdraw() functions lack reentrancy protection. Both functions interact multiple times with external contracts during execution, especially when adding or removing liquidity with the stableSwapPool. If the stableSwapPool contract is upgraded to malicious code, it could potentially reenter these functions during the calls to add_liquidity() or remove_liquidity(), leading to unintended modifications of the contract's state.

```
164 function deposit(uint256 usdtAmount, uint256 minLpAmount) external onlyActive {
165   require(usdtAmount > 0, "Invalid usdt amount");
166   uint256 expectLpAmount = getLpToMint(usdtAmount);
167   require(expectLpAmount >= minLpAmount, "Invalid min lp amount");
168
169   // 1. Transfer USDT to this contract
170   usdt.safeIncreaseAllowance(stableSwapPool, usdtAmount);
171   usdt.safeTransferFrom(msg.sender, address(this), usdtAmount);
172
173   // 2. Add USDT to PancakeStableSwapTwoPool
174   uint256 actualLpAmount = IERC20(lpToken).balanceOf(address(this));
175   IStableSwap(stableSwapPool).add_liquidity([0, usdtAmount], minLpAmount);
176   actualLpAmount = IERC20(lpToken).balanceOf(address(this)) - actualLpAmount;
177
178   require(actualLpAmount >= minLpAmount, "Invalid lp amount minted");
179
180   // 3. Update user's LP balance and LISTA reward, and distributor's LP total supply
181   _deposit(msg.sender, actualLpAmount);
182
183   // 4. Stake the received LP token to farming contract only if not in emergency mode
184   if (!emergencyMode) _stakeLp(msg.sender, actualLpAmount);
185
186   emit USDTStaked(lpToken, usdtAmount, actualLpAmount);
187 }
```

Listing 2.9: USDTLpListaDistributor.sol

```
195 function withdraw(uint256 lpAmount, uint256 minLisUSDAmount, uint256 minUSDTAmount) external {
196   require(lpAmount > 0, "Invalid LP amount");
197
198   // 1. Validate lisUSD and USDT amount
199   (uint256 expectLisUSDAmount, uint256 expectUSDTAmount) = getCoinsAmount(lpAmount);
200   require(minLisUSDAmount <= expectLisUSDAmount, "Invalid lisUSD amount");
201   require(minUSDTAmount <= expectUSDTAmount, "Invalid USDT amount");
202
203   // 2. Update user's LP balance and LISTA reward, and distributor's LP total supply
204   _withdraw(msg.sender, lpAmount);
205
206   // 3. Unstake LP token from farming contract if not in emergency mode
207   if (!emergencyMode) _unstakeLp(msg.sender, lpAmount);
208
209   // 4. Remove liquidity from PancakeStableSwapTwoPool
210   uint256 lisUSDAmountActual = lisUSD.balanceOf(address(this));
211   uint256 usdtAmountActual = usdt.balanceOf(address(this));
```

```
212
213     IStableSwap(stableSwapPool).remove_liquidity(lpAmount, [minLisUSDAmount, minUSDTAmount]);
214
215     lisUSDAmountActual = lisUSD.balanceOf(address(this)) - lisUSDAmountActual;
216     usdtAmountActual = usdt.balanceOf(address(this)) - usdtAmountActual;
217     require(lisUSDAmountActual >= minLisUSDAmount, "Invalid lisUSD amount received");
218     require(usdtAmountActual >= minUSDTAmount, "Invalid USDT amount received");
219
220     // 5. Transfer lisUSD and USDT to user
221     lisUSD.safeTransfer(msg.sender, lisUSDAmountActual);
222     usdt.safeTransfer(msg.sender, usdtAmountActual);
223
224     emit LpUnstaked(lpToken, usdtAmountActual, lisUSDAmountActual, lpAmount);
225 }
```

Listing 2.10: RewardDistributor.sol

Suggestion Add the `nonReentrant` modifier to prevent reentrancy attack risks.

2.3 Note

2.3.1 Potential centralization risk

Description In the current implementation, several privileged roles are set to govern and regulate system-wide operations (e.g., parameter setting, pause/unpause, and granting roles). Additionally, the function `emergencyWithdraw()` allows the `admin` role to withdraw all `LP` tokens from the farming contract into the contract. The `admin` also has the ability to arbitrarily update the `stakeVault` address. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to significant losses for users.

Feedback from the Project The `DEFAULT_ADMIN_ROLE` will be a `TimeLock` contract with a minimum 1 day delay.

2.3.2 Reward distribution mechanism and potential delay

Description The `USDTLpListaDistributor` contract distributes staking rewards periodically. Specifically, to avoid precision loss that could result in rewards becoming zero due to overly frequent reward claims in the farming contract, the contract introduces a global variable called `harvestTimeGap` to limit the frequency of reward claims. However, if this variable is set too large, it could lead to a temporary delay in reward distribution. After a reward distribution period ends, the rewards may pause until the time gap has passed. Rewards will only resume after this time gap has passed, either when a user passively triggers it (via a deposit or withdrawal action) or when the `harvest()` function is actively invoked. During this period, users will not receive rewards, which is an inherent trade-off of the mechanism.

Feedback from the Project The Bot currently performs the `harvest()` function every 12 hours on mainnet. Upon launch, the frequency will be adjusted to every 2 hours.

2.3.3 Expected reward distribution behavior during emergency mode

Description According to the design of `PancakeSwap Farming`, `CAKE` is distributed based on the amount of `LP` tokens actually staked in the `farming` contract. However, when the emergency mode is activated, `LP` tokens are withdrawn from the `farming` contracts and held in the contract. During this period, users who deposit can still receive the corresponding staking rewards, even though their deposited `LP` tokens are not accumulating rewards in the `farming` contracts at that time.

Feedback from the Project This design is intentional and aims to incentivize users to provide `USDT` liquidity by distributing farming rewards.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS