BLOCKSEC

Security Audit Report for LpTokenProvider

Date: April 3, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	LpTokenProvider

Version History

Version	Date	Description
1.0	April 3, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-04-22 05:23:40+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 LpTokenProvider[1] of Lista. Specifically, only the following contracts in the repository are included in the scope of this audit.

- contracts/dao/erc20LpProvider/ERC20LpTokenProvider.sol
- contracts/dao/interfaces/IERC20TokenProvider.sol
- contracts/dao/interfaces/ILpToken.sol
- contracts/dao/interfaces/IThenaErc20LpToken.sol
- contracts/dao/interfaces/IStableSwap.sol

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
LpTokenProvider	`Version 1`	`e7f7157db05f25631f348c3c8ed4bd47c3da0d1a`
	`Version 2`	`327eda26b14c87d9e5ecb79b430388a3529ffd8c`

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

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

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 \ 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.

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

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

Chapter 2 Findings

In total, we find **three** potential issues. Besides, we also have **three** recommendations and **four** notes.

- Medium Risk: 2
- Low Risk: 1
- Recommendation: 3
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Incorrect LP token mint address	DeFi Security	Fixed
2	Medium	Delayed synchronization of user LP token balances	DeFi Security	Confirmed
3	Low	Lack of pause check in function `syncUserLp()`	DeFi Security	Confirmed
4	-	Redundant balance check in function `_safeBurnLp()`	Recommendation	Confirmed
5	-	Lack of check in function `initialize()`	Recommendation	Fixed
6	-	Inconsistent validation logic in exchange rate and user LP rate settings	Recommendation	Fixed
7	-	Potential centralization risk	Note	-
8	-	Non-Transferable LP tokens only	Note	-
9	-	Potential risk of price manipulation	Note	-
10	-	Usage of exchangeRate in LP calculation	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect LP token mint address

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, users can deposit tokens into `lpProvidableDistributor` through the contract and specify a `delegatee` to receive `LP` tokens. However, the contract does not account for cases where users deposit tokens directly into `lpProvidableDistributor` without setting a `delegatee`. When such users invoke the function `syncUserLp()`, the contract retrieves their `LP` value via the function `getUserLpTotalValueInQuoteToken()` and mints `LP` tokens to the `delegatee`. Since the `delegatee` remains uninitialized (i.e., address(0)), the contract mints `LP` tokens to the zero address, which is incorrect.

```
226 function syncUserLp(address _account) external {
227   (bool rebalanced,) = _rebalanceUserLp(_account);
228   require(rebalanced, "already synced");
229 }
```

Listing 2.1: ERC20LpTokenProvider.sol

```
293 function _rebalanceUserLp(address account) internal returns (bool, uint256) {
294
295
296   // @dev this variable represent the latest amount of lpToken that user should have
297   //      user stakes LP(e.g. PCS stableSwap, Thena LP) which the LP binds with a certain amount
          of token0 and token1
298   uint256 userStakedTokenAmount = lpProvidableDistributor.getUserLpTotalValueInQuoteToken(
          account);
299
300
301   // ---- [1] Estimated LP value
302   // Total LP(Lista + User + Reserve)
303   uint256 newTotalLp = userStakedTokenAmount * exchangeRate / RATE_DENOMINATOR;
304   // User's LP
305   uint256 newUserLp = userStakedTokenAmount * userLpRate / RATE_DENOMINATOR;
306   // Reserve's LP
307   uint256 newReservedLp = newTotalLp - newUserLp;
308
309
310   // ---- [2] Current user LP and reserved LP
311   uint256 oldUserLp = userLp[account];
312   uint256 oldReservedLp = userReservedLp[account];
313
314
315   // LP balance unchanged
316   if (oldUserLp == newUserLp && oldReservedLp == newReservedLp) {
317       return (false, oldUserLp);
318   }
319
320
321   // ---- [3] handle reserved LP
322   if (oldReservedLp > newReservedLp) {
323       _safeBurnLp(lpReserveAddress, oldReservedLp - newReservedLp);
324       totalReservedLp -= (oldReservedLp - newReservedLp);
325   } else if (oldReservedLp < newReservedLp) {
326       lpToken.mint(lpReserveAddress, newReservedLp - oldReservedLp);
327       totalReservedLp += (newReservedLp - oldReservedLp);
328   }
329   userReservedLp[account] = newReservedLp;
330
331
332   // ---- [4] handle user LP and delegation
333   address holder = delegation[account];
334   if (oldUserLp > newUserLp) {
335       _safeBurnLp(holder, oldUserLp - newUserLp);
336   } else if (oldUserLp < newUserLp) {
337       lpToken.mint(holder, newUserLp - oldUserLp);
338   }
339   // update user LP balance as new LP
340   userLp[account] = newUserLp;
```

```
341
342
343     emit UserLpRebalanced(account, newUserLp, newReservedLp);
344
345
346     return (true, newUserLp);
347   }
```

Listing 2.2: ERC20LpTokenProvider.sol

Impact LP tokens are incorrectly minted to the zero address.

Suggestion Revise the logic to ensure the protocol can handle this scenario correctly.

2.1.2 Delayed synchronization of user LP token balances

Severity Medium

Status Confirmed

Introduced by Version 1

Description The internal function _rebalanceUserLp() is responsible for synchronizing users' LP token balances after actions such as deposit and withdrawal. This function ensures that the new LP value is correctly reflected by minting or burning LP tokens accordingly. Additionally, synchronization is required when a privileged role (MANAGER) updates the exchangeRate or userLpRate using functions setExchangeRate() and setUserLpRate().

However, synchronization is not automatically triggered when these rates are changed. Instead, it must be explicitly called, leading to potential delays. As a result, user LP balances may become outdated until a sync is manually initiated. This inconsistency can impact reward calculations, causing users to receive incorrect reward amounts based on stale LP values.

```
293 function _rebalanceUserLp(address account) internal returns (bool, uint256) {
294
295
296 // @dev this variable represent the latest amount of lpToken that user should have
297 //      user stakes LP(e.g. PCS stableSwap, Thena LP) which the LP binds with a certain amount of
          token0 and token1
298 uint256 userStakedTokenAmount = lpProvidableDistributor.getUserLpTotalValueInQuoteToken(account)
        ;
299
300
301 // ---- [1] Estimated LP value
302 // Total LP(Lista + User + Reserve)
303 uint256 newTotalLp = userStakedTokenAmount * exchangeRate / RATE_DENOMINATOR;
304 // User's LP
305 uint256 newUserLp = userStakedTokenAmount * userLpRate / RATE_DENOMINATOR;
306 // Reserve's LP
307 uint256 newReservedLp = newTotalLp - newUserLp;
308
309
310 // ---- [2] Current user LP and reserved LP
311 uint256 oldUserLp = userLp[account];
312 uint256 oldReservedLp = userReservedLp[account];
```

```
313
314
315 // LP balance unchanged
316 if (oldUserLp == newUserLp && oldReservedLp == newReservedLp) {
317     return (false, oldUserLp);
318 }
319
320
321 // ---- [3] handle reserved LP
322 if (oldReservedLp > newReservedLp) {
323     _safeBurnLp(lpReserveAddress, oldReservedLp - newReservedLp);
324     totalReservedLp -= (oldReservedLp - newReservedLp);
325 } else if (oldReservedLp < newReservedLp) {
326     lpToken.mint(lpReserveAddress, newReservedLp - oldReservedLp);
327     totalReservedLp += (newReservedLp - oldReservedLp);
328 }
329 userReservedLp[account] = newReservedLp;
330
331
332 // ---- [4] handle user LP and delegation
333 address holder = delegation[account];
334 if (oldUserLp > newUserLp) {
335     _safeBurnLp(holder, oldUserLp - newUserLp);
336 } else if (oldUserLp < newUserLp) {
337     lpToken.mint(holder, newUserLp - oldUserLp);
338 }
339 // update user LP balance as new LP
340 userLp[account] = newUserLp;
341
342
343 emit UserLpRebalanced(account, newUserLp, newReservedLp);
344
345
346 return (true, newUserLp);
347}
```

Listing 2.3: ERC20LpTokenProvider.sol

Impact Outdated `LP` balances can result in incorrect reward distributions, potentially causing financial discrepancies for users and the protocol.

Suggestion Timely invoke functions `syncUserLp()` and `bulkSyncUserLp()` to ensure `LP` balances remain accurate.

Feedback from the project The team will use an off-chain bot monitoring user's `clisBNB` position continuously, and invoke the `syncUserLp()` timely to ensure user's `clisBNB` balance is accurate.

2.1.3 Lack of pause check in function `syncUserLp()`

Severity Low

Status Confirmed

Introduced by Version 1

Description When the ERC20LpTokenProvider contract is paused, users are not allowed to deposit or withdraw LP tokens, which indirectly restricts new users from obtaining LP tokens. However, the user can directly deposit LP tokens into the lpProvidableDistributor contract and then invoke syncUserLp() to receive LP tokens. In this case, new users can still obtain LP tokens while the protocol is paused, which is incorrect.

```
226 function syncUserLp(address _account) external {
227     (bool rebalanced,) = _rebalanceUserLp(_account);
228     require(rebalanced, "already synced");
229 }
```

Listing 2.4: ERC20LpTokenProvider.sol

```
293 function _rebalanceUserLp(address account) internal returns (bool, uint256) {
294
295
296   // @dev this variable represent the latest amount of lpToken that user should have
297   //      user stakes LP(e.g. PCS stableSwap, Thena LP) which the LP binds with a certain amount
          of token0 and token1
298   uint256 userStakedTokenAmount = lpProvidableDistributor.getUserLpTotalValueInQuoteToken(
          account);
299
300
301   // ---- [1] Estimated LP value
302   // Total LP(Lista + User + Reserve)
303   uint256 newTotalLp = userStakedTokenAmount * exchangeRate / RATE_DENOMINATOR;
304   // User's LP
305   uint256 newUserLp = userStakedTokenAmount * userLpRate / RATE_DENOMINATOR;
306   // Reserve's LP
307   uint256 newReservedLp = newTotalLp - newUserLp;
308
309
310   // ---- [2] Current user LP and reserved LP
311   uint256 oldUserLp = userLp[account];
312   uint256 oldReservedLp = userReservedLp[account];
313
314
315   // LP balance unchanged
316   if (oldUserLp == newUserLp && oldReservedLp == newReservedLp) {
317       return (false, oldUserLp);
318   }
319
320
321   // ---- [3] handle reserved LP
322   if (oldReservedLp > newReservedLp) {
323       _safeBurnLp(lpReserveAddress, oldReservedLp - newReservedLp);
324       totalReservedLp -= (oldReservedLp - newReservedLp);
325   } else if (oldReservedLp < newReservedLp) {
326       lpToken.mint(lpReserveAddress, newReservedLp - oldReservedLp);
327       totalReservedLp += (newReservedLp - oldReservedLp);
328   }
```

```
329     userReservedLp[account] = newReservedLp;
330
331
332     // ---- [4] handle user LP and delegation
333     address holder = delegation[account];
334     if (oldUserLp > newUserLp) {
335         _safeBurnLp(holder, oldUserLp - newUserLp);
336     } else if (oldUserLp < newUserLp) {
337         lpToken.mint(holder, newUserLp - oldUserLp);
338     }
339     // update user LP balance as new LP
340     userLp[account] = newUserLp;
341
342
343     emit UserLpRebalanced(account, newUserLp, newReservedLp);
344
345
346     return (true, newUserLp);
347 }
```

Listing 2.5: ERC20LpTokenProvider.sol

Impact In the `paused` state, new users can still receive `LP` tokens, which is inconsistent with the intended design.

Suggestion Add a check to ensure that new users cannot receive `LP` tokens while the contract is `paused`.

Feedback from the project The `tokenProviderMode` will be turned on once `TokenProvider` is deployed and in use by the `ProvidableDistributor`, users are no-longer able to deposit/withdraw `LP` from the `ProvidableDistributor`.

2.2 Additional Recommendation

2.2.1 Redundant balance check in function `_safeBurnLp()`

Status Confirmed

Introduced by `Version 1`

Description The `_safeBurnLp()` function currently checks whether the parameter `amount` is greater than `availableBalance` before burning `LP` tokens from corresponding users. However, this check is unnecessary because the `amount` is always ensured to be less than or equal to `availableBalance`. In particular, function `_rebalanceUserLp()` ensures that the new `LP` balance is derived from the user-staked token value and predefined exchange rates. This can ensure that no user is required to burn more `LP` tokens than they own.

```
293 function _rebalanceUserLp(address account) internal returns (bool, uint256) {
294
295
296     // @dev this variable represent the latest amount of lpToken that user should have
297     //      user stakes LP(e.g. PCS stableSwap, Thena LP) which the LP binds with a certain amount
            of token0 and token1
```

```
298     uint256 userStakedTokenAmount = lpProvidableDistributor.getUserLpTotalValueInQuoteToken(
            account);
299
300
301     // ---- [1] Estimated LP value
302     // Total LP(Lista + User + Reserve)
303     uint256 newTotalLp = userStakedTokenAmount * exchangeRate / RATE_DENOMINATOR;
304     // User's LP
305     uint256 newUserLp = userStakedTokenAmount * userLpRate / RATE_DENOMINATOR;
306     // Reserve's LP
307     uint256 newReservedLp = newTotalLp - newUserLp;
308
309
310     // ---- [2] Current user LP and reserved LP
311     uint256 oldUserLp = userLp[account];
312     uint256 oldReservedLp = userReservedLp[account];
313
314
315     // LP balance unchanged
316     if (oldUserLp == newUserLp && oldReservedLp == newReservedLp) {
317         return (false, oldUserLp);
318     }
319
320
321     // ---- [3] handle reserved LP
322     if (oldReservedLp > newReservedLp) {
323         _safeBurnLp(lpReserveAddress, oldReservedLp - newReservedLp);
324         totalReservedLp -= (oldReservedLp - newReservedLp);
325     } else if (oldReservedLp < newReservedLp) {
326         lpToken.mint(lpReserveAddress, newReservedLp - oldReservedLp);
327         totalReservedLp += (newReservedLp - oldReservedLp);
328     }
329     userReservedLp[account] = newReservedLp;
330
331
332     // ---- [4] handle user LP and delegation
333     address holder = delegation[account];
334     if (oldUserLp > newUserLp) {
335         _safeBurnLp(holder, oldUserLp - newUserLp);
336     } else if (oldUserLp < newUserLp) {
337         lpToken.mint(holder, newUserLp - oldUserLp);
338     }
339     // update user LP balance as new LP
340     userLp[account] = newUserLp;
341
342
343     emit UserLpRebalanced(account, newUserLp, newReservedLp);
344
345
346     return (true, newUserLp);
347 }
348
349
```

```
350 /**
351 * @notice User's available lpToken might lower than the burn amount
352 *         due to the change of exchangeRate, ReservedLpRate or the value of the LP token
        fluctuates from time to time
353 *         i.e. userLp[account] might < lpToken.balanceOf(holder)
354 * @param holder lp token holder
355 * @param amount amount to burn
356 */
357 function _safeBurnLp(address holder, uint256 amount) internal {
358   uint256 availableBalance = lpToken.balanceOf(holder);
359   if (amount <= availableBalance) {
360       lpToken.burn(holder, amount);
361   } else if (availableBalance > 0) {
362       // existing users do not have enough lpToken
363       lpToken.burn(holder, availableBalance);
364   }
365 }
```

Listing 2.6: ERC20LpTokenProvider.sol

Suggestion Remove the redundant balance check in function `_safeBurnLp()` and directly burn `LP` tokens.

Feedback from the project This is because an address can be the `delegatee` of multiple users, we need to ensure the `amount` to burn will not exceeds user's actual `clisBNB` balance. So we decided to not make any change on this.

2.2.2 Lack of check in function `initialize()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract allows the privileged role `manager` to set the `exchangeRate` and `userLpRate` via the functions `setExchangeRate()` and `setUserLpRate()`. Both functions enforce a constraint ensuring that `userLpRate` is always less than `exchangeRate`.

However, the `initialize()` function does not perform this check when setting the initial values for `exchangeRate` and `userLpRate`. This inconsistency could lead to an invalid initial state where `userLpRate` is greater than or equal to `exchangeRate`, which would violate the intended constraints enforced elsewhere in the contract.

```
 96 function initialize(
 97     address _admin,
 98     address _manager,
 99     address _pauser,
100     address _lpToken,
101     address _token,
102     address _lpProvidableDistributor,
103     address _lpReserveAddress,
104     uint128 _exchangeRate,
105     uint128 _userLpRate
106 ) public initializer {
107     require(_admin != address(0), "admin is the zero address");
```

```
108     require(_manager != address(0), "manager is the zero address");
109     require(_pauser != address(0), "pauser is the zero address");
110     require(_lpToken != address(0), "lpToken is the zero address");
111     require(_token != address(0), "token is the zero address");
112     require(_lpProvidableDistributor != address(0), "_lpProvidableDistributor is the zero
            address");
113     require(_lpReserveAddress != address(0), "lpReserveAddress is the zero address");
114     require(_exchangeRate > 0, "exchangeRate invalid");
115     require(_userLpRate <= 1e18, "too big rate number");
116
117
118     __Pausable_init();
119     __ReentrancyGuard_init();
120     __UUPSUpgradeable_init();
121     __AccessControl_init();
122
123
124     // grant essential roles
125     _grantRole(DEFAULT_ADMIN_ROLE, _admin);
126     _grantRole(MANAGER, _manager);
127     _grantRole(PAUSER, _pauser);
128
129
130     token = _token;
131     lpToken = ILpToken(_lpToken);
132     lpProvidableDistributor = IERC20LpProvidableDistributor(_lpProvidableDistributor);
133     lpReserveAddress = _lpReserveAddress;
134     exchangeRate = _exchangeRate;
135     userLpRate = _userLpRate;
136
137
138     // approve max allowance in advance to save gas
139     IERC20(token).approve(_lpProvidableDistributor, type(uint256).max);
140 }
```

Listing 2.7: ERC20LpTokenProvider.sol

```
363 function setExchangeRate(uint128 _exchangeRate) external onlyRole(MANAGER) {
364     require(_exchangeRate > 0 && _exchangeRate >= userLpRate, "exchangeRate invalid");
365     exchangeRate = _exchangeRate;
366     emit ExchangeRateChanged(exchangeRate);
367 }
```

Listing 2.8: ERC20LpTokenProvider.sol

```
370 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
371     require(_userLpRate <= 1e18 && _userLpRate < exchangeRate, "userLpRate invalid");
372     userLpRate = _userLpRate;
373     emit UserLpRateChanged(userLpRate);
374  }
```

Listing 2.9: ERC20LpTokenProvider.sol

Suggestion Add a check in the function `initialize()` to ensure that `userLpRate` is less than `exchangeRate`.

2.2.3 Inconsistent validation logic in exchange rate and user LP rate settings

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The functions `setUserLpRate()` and `setExchangeRate()` have inconsistent validation rules regarding the relationship between `exchangeRate` and `userLpRate`. While function `setExchangeRate()` allows `exchangeRate` to be equal to `userLpRate`, function `setUserLpRate()` enforces `userLpRate` to be strictly less than `exchangeRate`.

```
363 function setExchangeRate(uint128 _exchangeRate) external onlyRole(MANAGER) {
364     require(_exchangeRate > 0 && _exchangeRate >= userLpRate, "exchangeRate invalid");
365     exchangeRate = _exchangeRate;
366     emit ExchangeRateChanged(exchangeRate);
367 }
```

Listing 2.10: ERC20LpTokenProvider.sol

```
370 function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {
371     require(_userLpRate <= 1e18 && _userLpRate < exchangeRate, "userLpRate invalid");
372     userLpRate = _userLpRate;
373     emit UserLpRateChanged(userLpRate);
374 }
```

Listing 2.11: ERC20LpTokenProvider.sol

Suggestion Revise the validation logic to ensure consistency.

2.3 Note

2.3.1 Potential centralization risk

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operations (e.g., parameter setting and pause/unpause). Additionally, the `admin` also has the ability to upgrade the implementation. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to losses for users.

2.3.2 Non-Transferable LP tokens only

Description The current implementation relies on the assumption that `LP` tokens are non-transferable. This ensures that user balances remain consistent and prevents external acquisition of `LP` tokens outside the controlled deposit process. If `LP` tokens were transferable, users could obtain them externally, bypassing system constraints and leading to inconsistencies in balance tracking and the rebalance mechanism.

2.3.3 Potential risk of price manipulation

Description The value returned by `getUserLpTotalValueInQuoteToken()` may reflect a spot price, which can be manipulated through large swaps. This could impact the `LP` amount received by users. In the event of significant market volatility or manipulation by malicious users through flash loans, it may lead to potential losses for the protocol.

Feedback from the project The team is aware, and plan to obtain a more legitimate price by introducing an oracle module in the future.

2.3.4 Usage of exchangeRate in LP calculation

Description The `getUserLpTotalValueInQuoteToken()` function reflects the `LP token` value, which increases due to swap fees in the `Pancake` pool. The additional multiplication by `exchangeRate` adjusts the `LP token` value relative to the quote token, requiring users to provide more quote tokens to mint `LP`. This is similar to over-collateralization, ensuring proper value and collateralization. The `exchangeRate` is set via a privileged function and its calculation is not visible within the contract.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS