BAIL security

BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

Lista DAO

slisBNB Provider

November 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
Website	lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/lista-token/blob/ea0490d890bb9ffa7d9d1c3851a7c3f91411326d/contracts/dao/VotingIncentive.sol https://github.com/lista-dao/lista-dao-contracts/blob/303c52c3973b2e9ac831bb63c2680a4b4523f8b6/contracts/ceros/provider/SlisBNBProvider.sol
Resolution 1	https://github.com/lista-dao/lista-token/blob/ea6719b1818fbc8aca763e124f816e3d3f2ad87b/contracts/dao/VotingIncentive.sol https://github.com/lista-dao/lista-dao-contracts/blob/bc21c3fd44e33160da245fede22717120eb9ac6b/contracts/ceros/provider/SlisBNBProvider.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1		
Medium	4	2		2
Low	8	1		7
Informational	17	5	1	11
Governance	1			1
Total	31	9	1	21

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

VotingIncentive

The VotingIncentive contract is a smart contract designed to encourage veLista token holders to participate in emission voting by offering them rewards.

It allows users to add incentives in the form of BNB or whitelisted ERC20 tokens, which are distributed to voters based on their voting weight for specific distributors and weeks.
Voters can claim these incentives proportionally to their contribution in the emission voting process.

Appendix: Incentive calculation

Once a week has surpassed and votes are immutable, users can claim their pro-rata share of provided incentives based on their provided VP and the overall VP (deducted by the admin VP).

> *userVP * incentives / (overallVP - adminVP)*

Appendix: Voting Mechanism

Users can allocate their veLista balance as voting weight across various distributors by specifying distributorIds and corresponding weights.
All users' votes are then aggregated to calculate the total weight for each distributor.. Voting is time-bound: regular users can vote up until a certain period before the next week (defined by ADMIN_VOTE_PERIOD), after which only users with the ADMIN_VOTER role can cast votes.
Once the voting period has passed and a new week starts, users can then claim their incentives based on the now immutable voting history.

Appendix: ListaVault

The ListaVault contract is responsible for managing and distributing LISTA token rewards to registered distributors. Administrators can deposit rewards for future weeks and set the percentage of emissions allocated to each distributor. Distributors claim these allocated rewards over time, and users can batch claim their rewards through the distributors.
Information about distributors is used within the VotingIncentive contract to determine for which distributors incentives can be distributed

Appendix: EmissionVoting

The EmissionVoting contract enables users who hold veLista tokens (locked LISTA balances) to influence the distribution of new LISTA emissions for the upcoming week. Users cast votes by allocating their veLista-derived voting weight across different distributors, specifying the distributor IDs and the amount of weight they wish to assign to each.

The contract collects these votes to determine the total weight allocated to each distributor, which then dictates how the new emissions are proportionally distributed among them.

This voting information within this contract is then used for voters to claim their share of incentives.

Appendix: Core Invariants

INV 1: Once a week has surpassed, historic votes must be immutable

INV 2: Admin VP must be excluded during allocation calculation

INV 3: userVotedDistributorIndex must return index +1 for the corresponding index of distributorId vote in Vote array

INV 4: Only whitelisted assets must be allowed as incentives

INV 5: Users cannot claim for future weeks

INV 6: Users cannot claim incentives more than once

INV 7: Incentives are distributed pro-rata based on voting weight

INV 8: Total claimable incentives must not exceed incentives added

INV 9: Only valid distributorIds must be used

INV 10: Users must only be allowed to claim incentives for weeks they have voted

Privileged Functions

- grantRole
- revokeRole
- whitelistAsset
- pause
- unpause
- setAdminVoter
- emergencyWithdraw

Issue_01	Incentive addition to disabled distributors will result in loss for incentivizer
Severity	**High**
Description	The addIncentivesBnb and addIncentives functions allow for the permissionless adding of incentives towards any future weeks. Incentives can be added to any existing distributor: *require(_distributorId > 0 && _distributorId <= vault.distributorId(), "Invalid distributorId");* A problem arises if incentives are added to a disabled distributor as this would mean users can never vote on this distributor and these incentives are effectively lost (until governance recovers these via emergencyWithdraw). The caller has effectively accidentally donated tokens without receiving any benefit from it.
Recommendations	Consider checking if a distributor is marked as disabled and disallow adding incentives in such a scenario. Another scenario where incentives are being added to a distributor which is currently enabled but will be disabled in future cannot be prevented. We recommend exercising caution if distributors are being disabled, keeping this scenario in mind.
Comments / Resolution	Resolved.

Issue_02	Dynamic adminVoter can inherently result in broken calculation
Severity	**Medium**

Description	Within the claim function, the allocated VP for a specific week and distributorId from the admin is fetched as follows:
	
	uint256 adminWeight = getRawWeight(adminVoter, _distributorId, _week);
	
	uint256 poolWeight = emissionVoting.getDistributorWeeklyTotalWeight(_distributorId, _week);
	uint256 usrWeight = getRawWeight(_user, _distributorId, _week);
	uint256 incentive = weeklyIncentives[_distributorId][_week][_asset];
	// If admin has voted, adjust user weight by removing admin weight from pool
	_amount = (usrWeight * incentive) / (poolWeight - _adminWeight);
	
	This practice is correct and necessary, as incentives are only meant for regular users and the lack of admin VP deduction would result in locked funds.
	
	However, this practice exposes some vulnerabilities:
	
	a) Change of admin voter within EmissionVoting contract without adjusting the adminVoter address
	
	b) More than one address with the ADMIN_VOTER role within the EmissionVoting contract
	
	In these scenarios, it can happen that the adminVoter address points to an invalid VP which would then either result in a loss of funds for claimers (if the adminVoter exposes insufficient VP, which then results in a higher divisor) or even worse in excess claimable funds with leaves some users with a loss (if the adminVoter exposes excessive VP, which then results in a smaller divisor).

Recommendations	Consider implementing a check within the setAdminVoter function which ensures that the new adminVoter has the corresponding role. (Similar as in the initialization) Furthermore, we recommend exercising general caution with the ADMIN_VOTER role.
Comments / Resolution	Resolved, such a check has been incorporated. Furthermore the team will ensure that there will always only be one address with the ADMIN_VOTER role.

Issue_03	Incorrect conditional check within batchClaim will prevent users from claiming
Severity	**Medium**
Description	The batchClaim function allows users to claim different assets from one week and one distributor, arbitrarily often. The function exposes a blunder within the following conditional statement: `if (!claimedIncentives[user][_params.distributorId][_params.week][_assets[j]]) continue;` In the scenario where an asset has not been claimed for a week and distributorId, the loop will continue, skipping the claim. This conditional statement should be inverted, as the loop should only skip the claim if the claim has already happened.
Recommendations	Consider inverting the statement.
Comments / Resolution	Resolved.

Issue_04	Native BNB transfer will not work if recipient is smart contract without fallback function
Severity	**Low**
Description	The receipt of native BNB within smart contracts is only possible if the contract exposes a fallback/receive function. In any other scenarios, the BNB receipt will simply fail. This will expose an issue for the claim function as under certain (above described) circumstances, claimers will not be able to claim native BNB.
Recommendations	Consider wrapping BNB to wBNB if the call reverts.
Comments / Resolution	Acknowledged.

Issue_05	Adding incentives after the voter period has passed is not beneficial
Severity	**Low**
Description	Currently, users can vote during a specific time window within a week. Once this window has passed, only the admin can vote. This exposes a small inconsistency with incentive adding, as the addition of incentives after the voter period (for the next week) has no further effect to incentivize users, thus it just means the caller will donate tokens.
Recommendations	Consider if it is desired to implement such a validation. Optionally, this issue can also be acknowledged.
Comments / Resolution	Acknowledged.

Issue_06	Memory caching for Votes array can become gas expensive
Severity	**Low**
Description	Within the getRawWeight function, all votes from one address during one week are casted as follows: *EmissionVoting.Vote[] memory votes = emissionVoting.getUserVotedDistributors(_account, _week);* If an address has voted for many different distributors, this will increase the Vote[] array, which then becomes gas-expensive during memory caching. In some very extreme instances, it can become so gas-expensive that it exceeds the block gas limit and the transaction reverts.
Recommendations	Within the current implementation, there is no trivial fix. A preliminary fix for this issue can be achieved by ensuring that the amount of different distributors becomes not excessively large (which lies in the hundreds range).
Comments / Resolution	Acknowledged, such a large amount of distributors will likely never be reached. The team is furthermore aware of this issue.

Issue_07	Incentives with no corresponding votes will result in temporarily stuck funds
Severity	**Low**
Description	A possible scenario is that a week and distributorId has received incentives but no voting has occurred for this week. This will result in these incentives being temporarily stuck within the contract until governance withdraws these funds.
Recommendations	Consider if this is seen as an issue, if yes, consider implementing functionality for incentive depositors to reclaim their incentives if no votes have happened. Alternatively, this issue can just be acknowledged.
Comments / Resolution	Acknowledged.

Issue_08	Permissionless claim can have trigger unexpected taxation events in certain jurisdictions
Severity	**Informational**
Description	The claim function allows claiming tokens for any address, via the _user parameter. In certain jurisdictions, it triggers a taxable event if a claim is received, even without triggering the transaction itself. This can result in unexpected scenarios for users in such a jurisdiction.
Recommendations	Consider if this is an acceptable risk, if not consider simply requiring (_user == msg.sender), which keeps compatibility with the batchClaim function.
Comments / Resolution	Ackowledged.

Issue_09	Contract is incompatible with certain ERC20 tokens
Severity	**Informational**
Description	The contract correctly handles transfer-tax tokens by implementing a before-after check. However, there are certain tokens that reduce their balance only on specific occasions. Such an example can be found here: https://github.com/bailsec/BailSec/blob/main/Bailsec%20-%20Ponzio%20The%20Cat%20Final%20Report.pdf For these tokens, the before-after check passes while the balance will decrease over time, resulting in insufficient tokens to honor all claims.
Recommendations	Consider implementing a strict due-diligence process before whitelisting tokens.
Comments / Resolution	Acknowledged.

Issue_10	Sub-optimal input validation within claim
Severity	**Informational**
Description	The claim function allows users to claim tokens using the following configuration: > distributorId > week > asset The week and distributorId parameters are validated as follows: *require(_week <= vault.getWeek(block.timestamp), "Invalid week");* *require(_distributorId > 0 && _distributorId <= vault.distributorId(), "Invalid distributorId");* The asset address is however not validated. While we could not identify an issue by providing an arbitrary asset parameter, 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 validating the asset parameter to ensure it has in fact allocated incentives for this specific distributor and week.
Comments / Resolution	Partially resolved, it is now checked that the asset has a corresponding incentive.

Issue_11	Division by zero possibility within calculateAmount
Severity	**Informational**
Description	The calculateAmount function executes the following arithmetic operation: *_amount = (usrWeight * incentive) / (poolWeight - _adminWeight);* This operation will revert if the divisor becomes zero, which is the case if no regular user has allocated VP to a distributor in one week.
Recommendations	Consider keeping this issue in mind.
Comments / Resolution	Resolved.

Issue_12	Lack of asset whitelisting check during claim
Severity	**Informational**
Description	The claim function allows for claiming any asset, without a check if an asset is whitelisted. While this prevents issues where assets would become accidentally unclaimable if whitelisting is removed, it limits governance restrictions.
Recommendations	Consider either keeping the current state or adding a whitelist check to the claim function. In the latter scenario, it has to be noted that this could result in some unclaimable tokens for users.
Comments / Resolution	Acknowledged.

Issue_13	Incentive addition accumulates dust
Severity	**Informational**
Description	Within the incentive addition, provided values are divided by the amount of weeks. This will result in some accumulated dust (in the range of a few wei) within the contract.
Recommendations	We do not recommend a change. This issue should be kept in mind.
Comments / Resolution	Acknowledged.

Issue_14	Missing init for UUPSUpgradeable
Severity	**Informational**
Description	The contract inherits from UUPSUpgradeable but does not call any initialization function related to it. It is best practice to call __UUPSUpgradeable_init.
Recommendations	Consider calling __UUPSUpgradeable_init during initialization.
Comments / Resolution	Resolved.

Issue_15	Lack of zero weeklyAmount check within addIncentivesBnb
Severity	**Informational**
Description	The addIncentivesBnb function has currently no check which ensures that weekyAmount is non-zero (contrary to that, addIncentives exposes this check). This can result in unexpected states.
Recommendations	Consider adding such a check.
Comments / Resolution	Acknowledged.

BaseTokenProvider

Disclaimer: This contract will only be audited in correspondence with the SlisBNBProvider. All virtual and overridden functions are excluded from the scope.

The following functions are overridden:

- _provideCollateral
- _burnLp
- _syncLp

The BaseTokenProvider contract forms the entry contract for users to provide tokens to receive slisBNB and release slisBNB, acting as an intermediary that facilitates the minting and burning of LpTokens, handles user delegations, and ensures synchronization between users' LP token balances and their actual holdings in the DAO based on the current exchangeRate and userLpRate.

The contract uses OpenZeppelin’s AccessControlUpgradeable contract for RBAC and is meant to be used as a proxy implementation using the UUPS pattern.

Appendix: Delegation Mechanism

The contract exposes two provide functions:

function provide(uint256 _amount)

function provide(uint256 _amount, address _delegateTo)

The second provide function allows users to determine a delegatee which receives the holderLpAmount of LpTokens.

This delegatee can freely use all tokens for any purpose. However, generally it is expected that the delegatee does not turn malicious and acts responsibly with the funds.

Furthermore, the delegateAllTo function allows for delegating all funds to an external address or removing all delegations and allocate it back to the own address.

Appendix: Release Mechanism

The release function allows a user to specify a desired amount of tokens to be withdrawn, withdraws these from the DAO address and burns the corresponding LpToken amount from the user, reserve and potentially delegatee.

Appendix: Core Invariants

INV 1: delegateAllTo must burn own amount and delegated amount

INV 2: delegateAllTo must delegate the full userLp amount to the own or external address

INV 3: delegateTo must always be set to a corresponding amount

INV 4: The provide function for delegations must increase userDelegation by the exact received lpHolderAmount

INV 5: delegateAllTo must not manipulate the reserve amount

INV 6: userLp must always be equal to delegated amount plus owned amount

INV 7: delegated amount must never exceed userLp

Appendix: Privileged Functions

- grantRole
- revokeRole
- liquidation
- daoBurn
- pause
- togglePause

Issue_16	Governance Privilege: Upgradeability
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The proxy admin can simply change the implementation of the proxy.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged

Issue_17	Provide does not prevent self-delegating
Severity	**Low**
Description	The provide function: *function provide(uint256 _amount, address _delegateTo)* Executes no check which ensures that _delegateTo is not msg.sender. Such an action can result in unexpected side-effects. 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 enforcing this requirement.
Comments / Resolution	Resolved.

Issue_18	Contract does not work with transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting. This issue is only raised as informational instead of high because the team is not planning to deploy the contract with token being a transfer-tax token. Furthermore, this would result in a revert during the DAO deposit
Recommendations	Consider not deploying the contract with a transfer-tax token as token.
Comments / Resolution	Acknowledged, such tokens will not be used.

Issue_19	daoBurn can result in stuck tokens within the DAO contract
Severity	**Informational**
Description	The withdraw function within the DAO contract has several different state possibilities. If under any circumstances it is not allowed for governance to directly withdraw funds from the DAO contract, it can result in locked funds if LpTokens from users are arbitrarily burned as this means the cross-contract interaction between BaseTokenProvider <-> DAO cannot be used to withdraw tokens.
Recommendations	Consider carefully elaborating such a scenario and the impact of calling daoBurn.
Comments / Resolution	Resolved, tokens are not burned anymore, the function now only syncs a position and has no effect at all.

SlisBNBProvider

The SlisBNBProvider contract extends the functionality of the BaseTokenProvider by introducing a variable exchange rate between the underlying token and the LP token. Unlike the base provider where the exchange rate is fixed at 1:1, this contract allows the exchange rate to be modified, offering greater flexibility in how tokens are exchanged for LpTokens.

Furthermore, a reserve mechanism is introduced which allocates a part of the minted LpTokens to the lpReserveAddress, based on the overall allocated LpToken amount and the userLpRate.

Before each interaction with the contract, the _syncLp function sync's a user's allocation by adjusting the current allocation based on an eventually adjusted exchangeRate and userLpRate.

The contract owner can change all rates at will.

Appendix: Exchange rate calculation

The contract exposes two rate variables:

a) exchangeRate: Determines how much LpTokens are corresponding to the amount of tokens
b) userLpRate: Determines how much LpTokens are allocated to userLp and how much to reserves

Upon each deposit, the contract calculates how much LpToken are received to userLp and how much are received to the reserves:

```
> uint256 lpAmount = _amount * exchangeRate / RATE_DENOMINATOR;
> uint256 holderLpAmount = lpAmount * userLpRate / RATE_DENOMINATOR;
> uint256 reservedLpAmount = lpAmount - holderLpAmount;
```

Upon each _syncLp call, the contract adjusts the allocated userLp and reserve amount based on the new exchangeRate and userLpRate.

Upon each withdrawal, the contract calculates how much LpTokens must be burned and how much will be taken from userLp and reserve:

```
> uint256 totalLpAmount = _amount * exchangeRate / RATE_DENOMINATOR;
> uint256 userPart = totalLpAmount * userLpRate / RATE_DENOMINATOR;
> uint256 reservePart = totalLpAmount - userPart;
```

Appendix: Sync Mechanism

The _syncLp function is triggered on the following occasions:

a) daoBurn
b) provide
c) delegateAllTo
d) release
e) syncUserLp

Thus, it is mandatory that this function is triggered before any user interaction. The purpose of this function is to adjust the slisBNB wallet balance and state within the contract whenever the ER or LPR has been adjusted via the changeExchangeRate and changeUserLpRate functions.

This function takes care of:

a) Reserve balance
b) Self balance
c) Delegated balance

The algorithm is as follows:

a) Determine how much LpToken a user should own based on the exchangeRate and the deposited funds to the DAO address
b) Determine the expected LpTokens to self based on the userLpRate
c) Determine the expected LpTokens to reserves based on the userLpRate
d) Cache the current reservePart
e) Cache the current userPart
f) Mint/burn to/from the reservePart to match the current reservePart with expected reservePart
g) Cache the current delegate amount

h) Calculate the new expected delegated amount:
   > expectUserLp * currentDelegateLp / userLp[user]
i) Mint/burn to/from the delegatee to match the expected delegated amount
j) Mint/burn to/from self to match the current self part with expected self part

Appendix: Reserved Mechanism

The userLpRate determines how much of the LpToken mint is going towards the user/delegatee and how much is going towards the reserve.

If a user receives 100e18 LpTokens and the userLpRate is 0.9e18, 90e18 tokens will go towards userLp and 10e18 tokens will go towards the reserve.

Appendix: LpToken

The LpToken contract can be found here:
https://bscscan.com/address/0xB0b84D294e0C75A6abe60171b70edEb2EFd14A1B
This token is minted/burned on every provide and release call as well as during the _syncLp process.

Appendix: DAO

The DAO contract code can be found here:

https://github.com/lista-dao/lista-dao-contracts/blob/303c52c3973b2e9ac831bb63c2680a4b4523f8b6/contracts/Interaction.sol

This contract will receive and custody all deposited tokens.

Appendix: Core Invariants:

INV 1: userLp[address] must always reflect accurately how much LP are allocated to an address

INV 2: userReservedLp[address] must always reflect accurately how much LP tokens from a user are allocated towards the reserve

INV 3: totalReservedLp must always reflect accurately how much LP tokens are aggregated towards reserves from all users

INV 4: userLp, userReservedLp, totalReservedLp must always be adjusted during LP sync

INV 5: userLp[address] must include delegated amounts

INV 6: _burnLp must always burn userReservedLp and userLp pro-rata

INV 7: _burnLp must first burn delegated amount

INV 8: lpReserveAddress must hold the accurate amount of LpTokens

INV 9: _burnLp must fully burn the corresponding user part and reserve part

INV 10: _syncLp must correctly adjust the delegated amount proportionally to userLp change

INV 11: During deposits, the received lpAmount must round down

INV 12: During withdrawals, the burned lpAmount must round up

INV 13: The contract must grant unlimited approvals to the DAO contract upon initialization

INV 14: The contract must define and assign all ROLES during initialization

INV 15: userLp[address] must accurately reflect LpToken amount to self and delegated amount

INV 16: exchangeRate must always be > 0

INV 17: lpReserveAddress change must mint all LpTokens to the new address

Privileged Functions

- grantRole
- revokeRole
- changeExchangeRate
- changeUserLpRate
- changeLpReserveAddress

Issue_20	Truncation within _burnLp allows for withdrawing tokens without burning corresponding LpToken
Severity	**Medium**
Description	The _burnLp function calculates the necessary amount of LpTokens to be burned in order to receive a corresponding amount of tokens: *uint256 totalLpAmount = _amount * exchangeRate / RATE_DENOMINATOR;* In the scenario where exchangeRate < RATE_DENOMINATOR, the truncation can be abused to burn zero amounts or lower amounts than expected. 99 * 1e16 / 1e18 = 0 > A user can withdraw 99 tokens for free if exchangeRate is 1e16 If the exchangeRate is 1e12 and the token has 6 decimals: 999999 * 1e12 / 1e18 > A user can withdraw for example 0.999 USDT for free The lower the exchangeRate and token decimals, the higher the damage. This method can be repeated to drain the contract.

	This issue is currently only rated as medium severity but can become even a high severity if the exchangeRate is unreasonably low and at the same time the token has low decimals.
Recommendations	Consider rounding up the calculation for totalLpAmount. It must be kept in mind that this will result in scenarios where users cannot withdraw their full deposited amount. Furthermore, Optionally, this issue can also be marked as resolved if it is by enforcing a requirement that the exchangeRate cannot be changed below a reasonable value (e.g. 1e17)
Comments / Resolution	Acknowledged. The team ensured that the ER is never set below 1e18.

Issue_21	Bypass of pausable feature via direct interaction with DAO
Severity	**Medium**
Description	The _syncLp function determines the overall amount of LpTokens which should be assigned to a user/reserves as follows: *uint256 expectAllLp = dao.locked(token, _account) * exchangeRate / RATE_DENOMINATOR;* The DAO contract allows anyone to deposit tokens on anyone's behalf: *function deposit(* *address participant,* *address token,* *uint256 dink* *) external* This means after a direct deposit interaction with the DAO, the expectAllLp function returns a larger value which then mints LpTokens to the user. This method can be used to receive LpTokens without invoking the provide function, by simply depositing directly in the DAO and then invoking syncUserLp. This call path allows users to successfully receive LpTokens even if the contract is in a paused state. Furthermore, this allows increasing tokens on behalf of other addresses, which may result in unexpected scenarios.
Recommendations	Consider only allowing a privileged role to invoke syncUserLp

Comments / Resolution	Acknowledged, direct interactions with the DAO contract will be prevented once this contract is deployed.

Issue_22	_burnLp does not burn delegated amount proportionally
Severity	**Low**

Description	The _burnLp function is invoked upon these occasions:
	a) daoBurn
	b) _withdrawLp (release)
	It first calculates the proportional reserve part to be burned:
	*uint256 totalLpAmount = _amount * exchangeRate / RATE_DENOMINATOR;*
	*uint256 userPart = totalLpAmount * userLpRate / RATE_DENOMINATOR;*
	uint256 reservePart = totalLpAmount - userPart;
	uint256 userTotalReserved = userReservedLp[_account];
	if (reservePart > 0) {
	lpToken.burn(lpReserveAddress, reservePart);
	userReservedLp[_account] -= reservePart;
	totalReservedLp -= reservePart;
	}
	And then burns the user part, including the delegation:
	if (delegation[_account].amount > 0) {
	uint256 delegatedAmount = delegation[_account].amount;
	uint256 delegateeBurn = userPart > delegatedAmount ? delegatedAmount : userPart;
	// burn delegatee's token, update delegated amount

	lpToken.burn(delegation[_account].delegateTo, delegateeBurn);
	delegation[_account].amount -= delegateeBurn;
	// burn delegator's token
	if (userPart > delegateeBurn) {
	_safeBurnLp(_account, userPart - delegateeBurn);
	}
	userLp[_account] -= userPart;
	
	A problem is that the delegation part is not taken proportionally. Instead an attempt is made to burn everything from the delegation part and then burn the eventual leftover from the user part.
	
	This issue is rated as medium severity because this will limit withdrawals. If for example the delegatee has transferred out a part of the funds, it will still attempt to burn all funds from the delegatee first instead of just burning the proportional amount, which can result in a DoS of the withdraw function even if the current balance of the delegatee added by the user self balance is sufficient to cover the request.

Recommendations	There are two solutions for this issue: a) Burning the delegated amount proportionally b) Allow the caller to determine how much should be burned from self and how much from the delegatee (during release) Any of these solutions would result in refactoring the withdrawal flow, which can possibly introduce unexpected side-effects. Optionally, this issue can also be acknowledged (if it is a justifiable design decision), which would then however limit withdrawal flexibility.
Comments / Resolution	Acknowledged, this is a design choice.

Issue_23	Faulty state during _safeBurnLp call via _burnLp
Severity	**Low**

Description	The _safeBurnLp function within the BaseTokenProvider contract determines the maximum available burn amount as follows:
	uint256 availableBalance = userLp[_account] - delegation[_account].amount;
	This is just a sanity check to ensure the contract does not accidentally burn more tokens than necessary. This scenario should however never occur.
	The _burnLp function invokes _safeBurnLp as follows:
	if (delegation[_account].amount > 0) {
	uint256 delegatedAmount = delegation[_account].amount;
	uint256 delegateeBurn = userPart > delegatedAmount ? delegatedAmount : userPart;
	// burn delegatee's token, update delegated amount
	lpToken.burn(delegation[_account].delegateTo, delegateeBurn);
	delegation[_account].amount -= delegateeBurn;
	// burn delegator's token
	if (userPart > delegateeBurn) {
	_safeBurnLp(_account, userPart - delegateeBurn);
	}
	The problem here is that the contract is in a faulty state, as the delegation has already been reduced:
	delegation[_account].amount -= delegateeBurn;
	But the userLp amount is still in its initial state.

	This will have the effect that the availableBalance value: *uint256 availableBalance = userLp[_account] - delegation[_account].amount;* Returns a higher value than the _account in fact owns. If for example at the beginning of the function the _account had a userLp value of 100e18 and a delegation of 50e18 and the function attempts to burn 100e18, after the burning of 50e18 from the delegation side, availableBalance will now erroneously return 100e18 because delegation has already been decreased, while it in fact only should return 50e18.
Recommendations	Since we could not find any edge-case where the correctness of the availableBalance is in fact necessary, this issue can be safely acknowledged.
Comments / Resolution	Acknowledged.

Issue_24	6 decimal token will result in incorrect LpToken amount
Severity	**Low**
Description	The LpToken amount for a corresponding amount of token is calculated as follows: *uint256 lpAmount = _amount * exchangeRate / RATE_DENOMINATOR;* If token has 6 decimals and exchangeRate is 1e18, this will result in a minted amount of LpToken with 6 decimals: 100e6 * 1e18 / 1e18 = 100e6 However, the LpToken is denominated with 18 decimals which means the depositor will only receive 0,0000000001 token instead of 100 tokens.
Recommendations	Consider either only using tokens with 18 decimals or up-scaling the amount. In the latter scenario, all state transitions must be carefully revalidated.
Comments / Resolution	Acknowledged, token will always have 18 decimals, as per team.

Issue_25	expectDelegateLp calculation will result in erroneous outcome if LPR is zero
Severity	**Informational**
Description	The initialize function currently allows for LPR to be set to zero: *require(_userLpRate <= 1e18, "too big rate number");* Under certain circumstances, this can be a valid setting (e.g. soft launch) A problem with this setting arises within the _syncLp function, more specifically in the way how expectDelegateLp is being calculated: *uint256 expectDelegateLp = userPart > 0 ? expectUserLp * currentDelegateLp / userPart : 0;* If userPart = 0, expectDelegateLp will just remain zero. Now as mentioned, this will become an issue if LPR is zero, as this would mean userLp[account] is zero and thus the delegated amount will not be adjusted. In the scenario where a user has delegated funds to another address during LPR = 0, these funds will just be allocated to the reserve. If now the LPR is changed to 1e18, these reserved funds should be allocated to the delegatee. However, due to the way the calculation is being executed, these funds are not being allocated to the delegatee but instead they are allocated to self: *uint256 expectDelegateLp = userPart > 0 ? expectUserLp * currentDelegateLp / userPart : 0;* ... if expectDelegateLp is erroneously zero ...

	uint256 expectUserSelfLp = expectUserLp - expectDelegateLp; … expectUserSelfLp becomes the full value … This will now result in a highly erroneous state as the user has effectively set a delegateTo but the delegated amount remains zero (while in fact it is desired that the delegatee will receive these funds).
Recommendations	Consider ensuring that LPR cannot be set to zero within the initialize function.
Comments / Resolution	Acknowledged, the issue severity has been decreased.

Issue_26	_burnLp does not reset delegatedTo if full delegated amount is withdrawn
Severity	**Informational**
Description	The _burnLp function may end up withdrawing the full delegated amount which sets delegation.amount to zero. In such a state, it is usually necessary for delegatedTo to be reset as well. However, currently that reset does not happen.
Recommendations	Consider resetting delegateTo if the _burnLp call results in delegatedAmount to become zero. Optionally, this issue can be acknowledged since we could not identify any downside from it.
Comments / Resolution	Acknowledged.

Issue_27	Precision loss within _provideCollateral
Severity	**Informational**

Description	The _provideCollateral function calculates how much LpTokens are corresponding to a specific amount of tokens and how much LpTokens should be allocated to the user and to the reserve:
	*uint256 lpAmount = _amount * exchangeRate / RATE_DENOMINATOR;*
	*uint256 holderLpAmount = lpAmount * userLpRate / RATE_DENOMINATOR;*
	Certain values can result in a precision loss based on exchangeRate and userLpRate. For example:
	lpAmount = 99 * 1e16 / 1e18
	lpAmount = 0
	Similar precision loss can happen within the _syncLp function.

Recommendations	Consider keeping reasonable exchange rates and implementing a slippage parameter upon provide, which ensures that lpAmount is always the desired value.
Comments / Resolution	Acknowledged.

Issue_28	Lack of proper initialization from inherited contracts
Severity	**Informational**

Description	The contract inherits the BaseTokenProvider which inherits multiple contracts which need proper initialization. Currently, the following contracts are initialized:
	__Pausable_init();
	__ReentrancyGuard_init();
	The AccessControlUpgradeable and UUPSUpgradeable contracts are not initialized.

Recommendations	Consider initializing these contracts.
Comments / Resolution	Resolved.

Issue_29	Potential overflow revert within _syncLp calculation
Severity	**Informational**
Description	The _syncLp function executes different arithmetic operations, for example: *uint256 expectAllLp = dao.locked(token, _account) * exchangeRate / RATE_DENOMINATOR;* If the locked token amount is unreasonably large and the exchangeRate as well, this could result in an overflow revert.
Recommendations	Consider implementing a reasonable upper limit for the ER.
Comments / Resolution	Acknowledged.

Issue_30	Contract combines flash minting and uint.max approval
Severity	**Informational**
Description	Upon initialization, the contract approves uint.max of token to the DAO address. Approvals are reduced whenever tokens are deposited to the DAO address via _provideCollateral: *dao.deposit(_account, token, _amount);* Since users can repetitively deposit and withdraw without any timelock, combined with the possibility that token can be a non-standard ERC20 token with > 18 decimals, the theoretical possibility for consuming approvals via flash minting is possible.
Recommendations	This is just a theoretical construct and is not likely to happen in practice as it would consume too many resources to execute. This issue was only added as an informational reminder of such a possibility. We do not recommend a change in the approval structure.
Comments / Resolution	Acknowledged.

Issue_31	Unused import
Severity	**Informational**
Description	The contract contains one or more imports which are completely unused. This will increase contract size for no reason and can confuse third-party reviewers: *import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";*
Recommendations	Consider removing this unused import.
Comments / Resolution	Resolved.