BAIL security

Lista DAO
Lista Lending - Update

FINAL REPORT

March ‘2025

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 Lending - Update
Website	Lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/moolah/tree/79f5dd0694bdba6732abe39752865579035e71e8 MoolahVault.sol SlisBNBProvider.sol BNBProvider.sol https://github.com/lista-dao/moolah/tree/5345086f1d200483fcce4a4a9dbca2eacf47795a Moolah.sol
Resolution 1	https://github.com/lista-dao/moolah/tree/cb18ad395e12b1ff3d612068843d3cd271f3029e

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	4	4			
Medium	8	2		6	
Low	10	4		6	
Informational	2	2			
Governance	1			1	
Total	25	12		13	

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 pose 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

Certain architectural choices and process-related aspects raised internal concern. We strongly recommend thorough reviews before future upgrades and improved documentation to support ongoing auditability.

Lista Lending

Differential Audit:

This contract is not fully audited but only the changes between the previous and new version have been audited. These changes are documented in the “Update” appendix.

Moolah is a decentralized lending protocol that enables users to supply, borrow, withdraw, and repay assets in various markets. The contract implements a permissionless money market system where users can supply assets as lenders, borrow against collateral, and participate in liquidations when positions become unhealthy.

At its core, Moolah functions as a lending pool where users can deposit tokens (supply) to earn interest and borrow assets against their collateral. The contract utilizes a shares-based accounting system to track users' positions and manage interest accrual. Each market is defined by parameters including loan token, collateral token, oracle address, interest rate model (IRM), and liquidation loan-to-value (LLTV) ratio, which together determine the borrowing capacity.

The protocol incorporates robust risk management features, including a whitelist-based liquidation mechanism that allows designated accounts to liquidate unhealthy positions when borrowers fall below required collateralization thresholds. When liquidations occur, liquidators can seize collateral at a discount (liquidation incentive) in exchange for repaying a portion of the borrower's debt, protecting the protocol against bad debt accumulation.

Appendix: Updates

The following changes have been introduced since the previous audit:

1. Operator can now add and remove providers:
   - **addProvider()** - only the manager can assign a provider for a specific token.
   - **removeProvider()** - only the manager can reset the assigned provider for a token.

2. Providers integrated into the following functions:
   - **borrow()** — Users can initiate borrowing either directly or via a designated provider. If a provider is used for the specified loan token, the receiver must match the provider address.
   - **supplyCollateral()** — When a provider is registered for a given collateral token, collateral deposits must be routed through that provider. Direct calls by users to supplyCollateral() are only allowed if no provider is set.
   - **withdrawCollateral()** — If a provider is assigned to the collateral token used in the market, users are restricted from calling withdrawCollateral() directly. Instead, the withdrawal must be initiated and received by the provider.
   - **liquidate()** — During liquidation, if a provider is configured for the collateral token, a liquidate() call is forwarded to the provider's own liquidate() function after completing internal state updates.

3. The nonReentrant modifier was removed from the flashLoan() function.

Core Invariants:

INV 1: Total borrowed assets must never exceed total supplied assets in a market.

INV 2: A user cannot create or modify a position in a way that would result in it becoming unhealthy.

INV 3: Shares must accurately reflect proportional ownership.

INV 4: Interest is accrued before every position-changing action to reflect real-time debt growth.

INV 5: Only governance-enabled IRMs and LLTVs can be used to create new markets.

INV 6: Flash loans must be repaid within the same transaction or the call reverts.

Privileged Functions

- enableIRM
- enableLltv
- setFee
- setFeeRecipient
- addLiquidationWhitelist
- removeLiquidationWhitelist
- pause
- unpause
- addProvider
- removeProvider

Issue_01	MANAGER could add a malicious or broken provider
Severity	**Governance**
Description	The MANAGER role can add providers for each token using addProvider(). The new provider address that is assigned to a token will be able to invoke the following functions: 1. borrow() - borrow on behalf of any user and receive the borrowed funds. 2. withdrawCollateral() - withdraw collateral on behalf of any user and receive the withdrawn collateral. 3. liquidate() - a broken provider might revert the liquidate call and prevent liquidations.
Recommendations	Consider ensuring any new provider will work as expected.
Comments / Resolution	Acknowledged.

Issue_02	Users can’t directly supply/withdraw WBNB from Moolah
Severity	**Low**
Description	When admin sets BNBProvider for WBNB markets then code won’t allow users to directly supply/withdraw WBNB from Moolah markets directly. As a result, if users want to supply WBNB, they need to first convert WBNB to BNB and then supply through the BNBProvider contract. Also some users may want to receive their WBNB when withdrawing their collateral but they would always receive it as BNB token which they need to wrap it. The code shouldn’t allow users to directly supply/withdraw collateral when collateral is slisBNB but this limitation shouldn’t be applied to markets where WBNB is collateral which WBNBProvider is set for them.
Recommendations	Consider enabling users to directly interact with markets that accept WBNB as collateral. To support this, the implementation should differentiate between BNBProvider and SlisBNBProvider, ensuring that only the BNBProvider is permitted to access such markets directly. In contrast, direct user interaction with markets where slisBNB is used as collateral should remain restricted
Comments / Resolution	Acknowledged.

Issue_03	Read-only reentrancy in the liquidation transaction
Severity	**Low**

Description	In markets where slisBNB is used as the collateral token, a read-only reentrancy issue may arise during liquidation. Specifically, the system state becomes inconsistent between the updated collateral and the corresponding lpBalance of the borrower at a critical point in the liquidation flow.
	
	During a liquidation transaction, the borrower's collateral is decreased before their lpBalance is updated. This occurs just prior to invoking the external callback onMoolahLiquidate():
	
	function liquidate(
	...
	) external whenNotPaused nonReentrant returns (uint256, uint256) {
	...
	position[id][borrower].collateral -= seizedAssets.toUint128();
	...
	if (data.length > 0) IMoolahLiquidateCallback(msg.sender).onMoolahLiquidate(repaidAssets, data);
	...
	if (providers[marketParams.collateralToken] != address(0)) {
	IProvider(providers[marketParams.collateralToken]).liquidate(id, borrower);
	}
	return (seizedAssets, repaidAssets);
	}
	
	At this stage, an external contract could exploit the temporary inconsistency to perform a read-only reentrancy attack—i.e., read and rely on the unsynchronized state.

	Execution flow: - Moolah.liquidate() is called. - The contract reduces the borrower's collateral, but does not update lpBalance yet. - External call to msg.sender.onMoolahLiquidate() is made. - During this call, the attacker performs a read-only reentrancy, accessing an unsynced lpBalance that still reflects the pre-liquidation state. - Provider.liquidate() is invoked with an inconsistent internal state.
Recommendations	Consider calling onMoolahLiquidate() after the Provider.liquidate() call.
Comments / Resolution	Resolved by following the recommendation.

Issue_04	addProvider() can be used to change the existing provider
Severity	**Informational**
Description	addProvider() performs the following check: *function addProvider(address token, address provider) external onlyRole(MANAGER) {* *require(providers[token] != provider, ErrorsLib.ALREADY_SET);* However, rather than verifying that a provider has not already been set, this check only ensures that the new provider is not identical to the existing one. As a result, the function may unintentionally allow overwriting an existing provider, effectively acting as an update rather than strictly adding a new one.
Recommendations	Consider changing the check to: *require(providers[token] != address(0), ErrorsLib.ALREADY_SET);*
Comments / Resolution	Resolved by following the recommendation.

Issue_05	addProvider() does not validate if the provider is correct for the token
Severity	**Informational**

Description	addProvider() performs the following checks:
	
	function addProvider(address token, address provider) external onlyRole(MANAGER) {
	require(token != address(0), ErrorsLib.ZERO_ADDRESS);
	require(provider != address(0), ErrorsLib.ZERO_ADDRESS);
	require(providers[token] != provider, ErrorsLib.ALREADY_SET);
	However, it does not verify whether the provider's associated token matches the token it is being assigned to.

Recommendations	Consider checking if the provider’s token is the same as the token passed in the addProvider() call.
Comments / Resolution	Resolved.

MoolahVault

Differential Audit: This contract is not fully audited but only the changes between the previous and new version have been audited. These changes are documented in the “Update” appendix.

MoolahVault is an ERC4626-compliant vault that enables users to deposit assets into Moolah. The vault acts as an intermediary layer between users and the Moolah protocol, allowing for simplified asset management, yield optimization, and risk diversification.

The contract implements a sophisticated allocation system that distributes deposited assets across multiple Moolah markets according to configurable caps and priorities. This is managed through supply and withdraw queues, which determine the order in which markets receive deposits or provide withdrawals. The vault's architecture allows for granular control over market exposure while abstracting away the complexity from end users who simply interact with the vault's deposit and withdrawal functions.

A key feature of MoolahVault is its role-based access control system, which defines distinct responsibilities: Managers can adjust fee parameters and set recipients, Curators can manage market configurations and removal processes, and Allocators can rebalance assets across markets.

The contract includes a fee mechanism that accrues performance fees based on the interest earned from Moolah markets. These fees are calculated as a percentage of the total interest earned and minted as vault shares to a designated fee recipient.

Appendix: Updates

The following changes have been introduced since previous audit:

1. A new BOT role has been introduced to support automated or off-chain controlled rebalancing. The role is managed by the ALLOCATOR, who can grant or revoke it using the following functions:

    - **setBotRole()** — Allows the allocator to assign the BOT role to a specified address.
    - **revokeBotRole()**— Enables the allocator to remove the BOT role from an address.

        Any address with the BOT role is authorized to call the reallocate() function, allowing it to perform market reallocations on behalf of the vault.

2. Support for external providers has been added, enabling delegated access to vault operations through two dedicated functions:

    - withdrawFor() – Allows a registered provider to withdraw a specified amount of assets on behalf of a user. The provider receives the withdrawn assets, and the corresponding shares are burned from the owner's balance.
    - redeemFor() – Enables a provider to redeem a specified number of shares on behalf of a user. The provider receives the resulting assets, and the user's share balance is reduced accordingly.

Appendix: Invariants

INV 1: On reallocate total asset withdrawn must equal total asset supplied

INV 2: Only 0 supply cap market is removable from vault

INV 3: All user provided asset must be supplied to an underlying market

INV 4: Supply and withdraw queue cannot exceed the maximum queue length (30)

Privileged Functions

- setSkimRecipient
- setFee
- setFeeRecipient
- setCap
- setMarketRemoval
- setSupplyQueue
- updateWithdrawQueue
- reallocate

Issue_6	Missing slippage parameters when interacting with MoolahVault
Severity	**Low**
Description	The MoolahVault contract allows users to add/remove tokens through the deposit, mint, withdraw, and redeem functions. These operations involve conversions between tokens and vault shares However, these functions currently lack slippage protection. This means users may receive fewer assets or shares than expected due to changes in the MoolahVault exchange rate, which can fluctuate over time due to interest accrual or realized bad debt. Without slippage parameters, users are exposed to value discrepancies when interacting with the vault, especially during volatile conditions or periods of rapid change in the share-to-asset ratio.
Recommendations	Consider adding slippage parameters to the deposit, mint, withdraw, and redeem functions in MoolahVault. This will ensure users can specify minimum acceptable amounts, protecting them from receiving less value than expected.
Comments / Resolution	Acknowledged.

SlisBNBProvider

The SlisBNBProvider contract integrates slisBNB tokens into the Moolah CDP (Collateralized Debt Position) system, allowing users to supply slisBNB as collateral. Upon depositing, users are issued clisBNB—an LP token that represents their share of the collateral and serves as a receipt of deposit. This token is minted based on a configurable userLpRate, which determines the proportion of the underlying converted value allocated to the user.

Users may optionally delegate their clisBNB to another address (the delegatee), which then becomes the sole holder of all LP tokens associated with that user's deposits. All clisBNB minting, burning, and transfers are handled on the delegatee's balance. When users withdraw collateral or are subject to liquidation, their corresponding clisBNB is burned from the delegatee's balance. The actual clisBNB holdings of the delegatee are dynamically updated through a synchronization process that reflects changes in the delegator's (onBehalfOf) collateral position within the Moolah system.

Appendix: SlisBNB

slisBNB is a liquid staking token (LST) issued on BNB Smart Chain by Lista DAO. It represents staked BNB and accrues staking rewards over time, increasing in value relative to BNB. When users stake BNB through the protocol, they receive slisBNB, which remains transferable and usable in smart contracts while the underlying BNB remains locked.

The SlisBNBProvider contract allows users to deposit slisBNB into the Moolah protocol as collateral, issuing clisBNB in return.

Appendix: ClisBNB

clisBNB is a non-transferable token issued by Lista DAO that represents BNB collateral deposited into the protocol's Moolah CDP system. It serves as an internal accounting mechanism to track user deposits and is minted at a 1:1 ratio to the amount of BNB supplied. clisBNB is burned when the corresponding collateral is withdrawn or liquidated. It cannot be transferred between accounts but may be delegated at the time of deposit to a specified address for purposes such as participating in Binance Launchpool events or borrowing lisUSD.

Appendix: User Collateral Synchronization via _syncPosition()

Whenever the user’s collateral in a CDP position changes, the _syncPosition() function is invoked. This function ensures that the user’s (or their delegatee’s) clisBNB balance is dynamically adjusted to reflect the current collateral state in the Moolah protocol. It performs the following steps to synchronize the user's clisBNB balance with their updated collateral:

1. It retrieves the user's current collateral from their position and compares it to the previously recorded value in the userMarketDeposit mapping.
2. Based on the difference, the userTotalDeposit is updated to reflect the net change in collateral.
3. The user’s total deposited amount (userTotalDepositAmount) is then converted from slisBNB to BNB using the StakeManager.
4. A new LP token amount (newUserLp) is calculated using the current userLpRate defined by the Provider.
5. Finally, the new LP token values are compared with the previous ones to determine the necessary amount to mint or burn from the user's and MPC wallets' balances, ensuring consistency across the system.

Core Invariants:

INV 1: Sender must be authorized to withdraw collateral.
INV 2: The amount minted to a wallet cannot exceed the MPCWallet cap.
INV 3: removeMPCWallet() cannot remove wallets which have a non-zero balance.

Privileged Functions

- setUserLpRate
- setMpcWalletCap
- removeMPCWaller
- addMPCWallet
- _authorizeUpgrade

Issue_7	Users can mint unbacked clisBNB by syncing their internal accounting using a market ID that uses a different collateral than SlisBNB on Moolah
Severity	**High**
Description	In SlisBNBProvider::syncUserLp(), the caller specifies the marketID that will be used to read the user's collateral to update his internal accounting to rebalance his clisBNB balance. However, Id can be any marketID on Moolah, regardless if the collateralToken of the market is SlisBNB or a different asset. This can be exploited to mint unbacked clisBNB tokens by leveraging all the markets where the user has collateral, specially for collateralTokens that are worth a fraction of what SlisBNB is worth. Consider the following scenario: 1. A user supplies collateral, on a MarketID that uses DAI (or other stablecoin w/ 18 decimals) as the collateralToken. 2. The user calls syncPosition() specifying the marketID of the market where collateral was supplied on step 1. 3. MOOLAH.position(id, account).collateralwill return the amount of DAI supplied as collateral on that market. 4. The user's LP accounting will be rebalanced using the value read on step 3. 5. As a result, clisBNB will be minted for the collateral supplied on a market that is using a different asset than SlisBNB. As a result, the user will be able to mint unbacked clisBNB by using his collateral supplied on a market that does not use SlisBNB as collateral.
Recommendations	Consider validating that the collateralToken of the specified marketId is actually SlisBNB.
Comments / Resolution	Resolved following recommendation. Added token validation in _syncPosition() **Disclaimer: The initial commit assigned providers per token. During the resolution phase, this was updated to assign providers per market ID. The change was reviewed only within the time-constrained resolution round.**

Issue_8	Anyone can reset the delegation of other users
Severity	**High**

Description	The supplyCollateral() function performs the following change of the delegatee:
	`// get current delegatee`
	`  address oldDelegatee = delegation[onBehalf];`
	`  // burn all lpToken from old delegatee`
	`  if (oldDelegatee != onBehalf && oldDelegatee != address(0)) {`
	`    _safeBurnLp(oldDelegatee, userLp[onBehalf]);`
	`    // clear user's lpToken record`
	`    userLp[onBehalf] = 0;`
	`  }`
	However, because the protocol permits collateral to be supplied on behalf of any user, it also implicitly allows third parties to override delegation settings. Specifically, any external actor can reset a user's delegatee address by making a collateral deposit on their behalf even with 1 wei.
	Consider the following scenario:
	1) UserA supplies slisBNB as collateral and explicitly delegates their clisBNB to UserB by calling delegateAllTo(UserB).
	2) A malicious actor (or any third party) then calls supplyCollateral on behalf of UserA. During this process, the contract logic resets the delegatee back to UserA, effectively revoking the prior delegation to UserB.

Recommendations	Consider leaving the delegatee address unchanged when supplyCollateral() is called, instead of automatically resetting it to onBehalf.
Comments / Resolution	Resolved following recommendation. Removed delegation change logic from supplyCollateral().

Issue_9	Attacker can mint unbacked clisBNB
Severity	**High**

Description	In SlisBNBProvider::withdrawCollateral(), msg.sender is passed as the input to _syncPosition():
	`function withdrawCollateral(MarketParams memory marketParams, uint256 assets, address onBehalf, address receiver)`
	`    external`
	`  {`
	`   ...`
	`    // withdraw from distributor`
	`    MOOLAH.withdrawCollateral(marketParams, assets, onBehalf, address(this));`
	`    // rebalance user's lpToken`
	`    _syncPosition(marketParams.id(), msg.sender);`
	However, msg.sender can be any address authorized to manage the borrower's position via the Moolah protocol. As a result, instead of updating the LP token balance for the actual borrower (onBehalf), _syncPosition() mistakenly updates the LP balance for the caller (msg.sender).
	This can be exploited to mint unbacked clisBNB tokens due to the logic in _safeBurnLp(), which attempts to burn only up to the available LP token balance of the target address:
	`function _safeBurnLp(address holder, uint256 amount) internal {`
	`    uint256 availableBalance = lpToken.balanceOf(holder);`
	`    if (amount <= availableBalance) {`
	`      lpToken.burn(holder, amount);`
	`    } else if (availableBalance > 0) {`
	`      // existing users do not have enough lpToken`
	`      lpToken.burn(holder, availableBalance);`
	`    }`
	Consider the following scenario:
	1) An attacker opens a position and supplies collateral, receiving clisBNB in return.
	2) The attacker authorizes addressA to act on their behalf.

	3) addressA calls withdrawCollateral(). During this call: -The borrower's collateral is withdrawn. - _syncPosition() is called with msg.sender (addressA) instead of the borrower. 4) Since addressA has no clisBNB, _safeBurnLp() burns 0 tokens. As a result, the attacker's actual LP balance remains untouched, effectively allowing them to withdraw collateral while retaining their clisBNB—creating unbacked tokens.
Recommendations	Consider using onBehalf when calling _syncPosition()
Comments / Resolution	Resolved following recommendation.

Issue_10	ClisBNB doesn't keep 1:1 ratio with BNB
Severity	**Medium**
Description	According to the documentation, clisBNB is intended to maintain a 1:1 ratio with the user's BNB collateral. However, in the current implementation, the clisBNB amount is only calculated at the time of interaction (e.g., deposit or withdrawal). Outside of these interactions, the LP token balance remains unchanged, and it is not updated to reflect changes in the SnBNB conversion rate. As a result, the 1:1 ratio is not consistently maintained. Because of this design, each user's lpBalance is effectively calculated using the SnBNB rate at the time of their most recent interaction. Consequently, clisBNB balances can become misaligned with the actual value of the user's BNB collateral. This leads to an inconsistency where some users hold disproportionately high clisBNB balances relative to their current collateral value, resulting in unfair advantages and potential over-rewarding.
Recommendations	Consider updating the documentation or changing the implementation to a rebase token so that lpBalance always shows the current 1:1 ratio with BNB for all users.
Comments / Resolution	Acknowledged.

Issue_11	Users who deposited into a slisBNB market before the provider was set can prevent liquidations
Severity	**Medium**
Description	An attacker can supply collateral to a Moolah market that accepts slisBNB before the SlisBNBProvider is configured. In this scenario, the attacker holds slisBNB as collateral but has not received any clisBNB. As more users interact with the SlisBNBProvider, the minting cap on the Binance MPC wallets may eventually be reached, preventing additional clisBNB from being minted. If the attacker's position becomes unhealthy at this point, it cannot be liquidated. Attempting to liquidate the position would require minting clisBNB to the attacker, since some collateral remains. However, due to the cap being reached, the minting fails, and the transaction reverts. This effectively blocks the liquidation. As a result, the attacker's position cannot be liquidated unless one of the following conditions is met: ● Other users withdraw enough collateral to bring the minting capacity below the cap. ● The position accrues bad debt, and the remaining collateral becomes zero, allowing liquidation to proceed without minting. In the second case, the position is liquidated with bad debt, leading to losses for other users in the same Moolah market.
Recommendations	Consider calling bulkSyncUserLp for all pre-existing positions in markets using slisBNB as collateral when SlisBNBProvider is added to the Moolah protocol.
Comments / Resolution	Acknowledged.

Issue_12	DoS on setUserLpRate() due to exchange rate check
Severity	**Medium**
Description	The userLpRate variable determines the fee, in clisBNB, that the protocol earns when users deposit slisBNB as collateral on Moolah. Admins can update this fee using the setUserLpRate function. However, this function includes a check that requires the new userLpRate to be less than the current exchangeRate: *function setUserLpRate(uint128 _userLpRate) external onlyRole(MANAGER) {* *require(_userLpRate <= 1e18 && _userLpRate <= exchangeRate, "userLpRate invalid")* The issue is that exchangeRate is a variable that is never updated or used elsewhere in the contract. Since its value remains zero, the setUserLpRate function will only accept a new rate of zero. This effectively blocks any updates to the protocol fee, resulting in a denial of service.
Recommendations	Consider removing the unused exchangeRate variable from the contract. Its presence causes a denial of service on setUserLpRate and serves no functional purpose.
Comments / Resolution	Resolved following recommendation.

Issue_13	SlisBNBProvider does not support any of the Moolah callbacks
Severity	**Medium**
Description	The following functions support callback logic in Moolah: • repay() - triggers a callback to the msg.sender via onMoolahRepay() • supplyCollateral() - triggers a callback to msg.sender via onMoolahSupplyCollateral() However, these callbacks are no longer applicable when interacting via the SlisBNBProvider. Additionally, users can only supply collateral through the provider due to the following require statement, which restricts direct interaction: *if (providers[marketParams.collateralToken] != address(0)) {* *require(msg.sender == providers[marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);* *}* As a result, users can no longer utilize the onMoolahSupplyCollateral() callback for markets where collateralToken == slisBNB, since collateral must be supplied via the SlisBNBProvider. Similarly, the onMoolahRepay() callback becomes inaccessible when repaying through the slisBNBProvider (it may be used if repay() is called directly on Moolah).
Recommendations	Consider forwarding the onMoolahRepay() and onMoolahSupplyCollateral() callbacks to the sender in SlisBNBProvider.
Comments / Resolution	Acknowledged.

Issue_14	Users may be unable to supply collateral to their positions
Severity	**Medium**

Description	Due to the following check, users in markets where collateralToken == slisBNB can only supply collateral through the slisBNBProvider:
	function supplyCollateral(MarketParams memory marketParams, uint256 assets, address onBehalf, bytes calldata data)
	{
	if (providers[marketParams.collateralToken] != address(0)) {
	require(msg.sender == providers[marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);
	}
	However, users may be unable to supply collateral if all MPC wallets have reached their caps, as enforced by the following require statement:
	function _mintToMPCs(uint256 amount) internal {
	require(leftToMint == 0, "Not enough MPC wallets to mint");
	}
	As a result, users who urgently need to supply collateral to protect the health of their positions may be unable to do so.

Recommendations	If this limitation is intentional, clearly communicate to users that depositing slisBNB may sometimes fail when MPC caps are reached. If not intended, consider allowing users to call Moolah::supplyCollateral directly as a fallback. This would let them add collateral without minting clisBNB.
Comments / Resolution	Acknowledged.

Issue_15	removeMPCWallet can be griefed by frontrunning
Severity	**Medium**
Description	When admins attempt to remove an MPC wallet from SlisBNBProvider, they use the removeMPCWallet function. This function requires the specified wallet to have a zero balance of clisBNB. MPC wallets are used in order: minting operations consume wallets from first to last, while burning operations go from last to first. Since admins cannot reorder the list of MPC wallets, removing a wallet near the beginning of the list becomes extremely difficult. These wallets are unlikely to be emptied naturally through normal usage. Additionally, an attacker can grief this process by frontrunning the call to removeMPCWallet. By supplying just 1 wei of collateral, the attacker causes a clisBNB mint to the targeted wallet, giving it a non-zero balance and reverting the admin's transaction.
Recommendations	Consider adding a function that allows admins to reorder the MPC wallet list. This would enable them to move a specific wallet to the end of the array, where it can be emptied over time through withdrawals and then safely removed.
Comments / Resolution	Acknowledged.

Issue_16	setMPCWalletCap can be griefed by frontrunning
Severity	**Low**
Description	Admins use the setMPCWalletCap function to adjust the cap of an MPC wallet. If the new cap is lower than the wallet’s current balance, the contract attempts to mint clisBNB to other MPC wallets to rebalance the excess. However, an attacker can frontrun this call by supplying a large amount of slisBNB as collateral. This fills all available MPC wallet caps, making it impossible for the contract to redistribute the excess clisBNB and preventing the cap reduction from succeeding. This allows an attacker to repeatedly block attempts to lower MPC wallet caps whenever the new cap is set below the current balance.
Recommendations	If this is not the intended behavior, consider adding a boolean flag to setMPCWalletCap that temporarily ignores the other wallets' caps and allows minting only the amount necessary to complete the cap reduction.
Comments / Resolution	Acknowledged.

Issue_17	Users cannot set a delegate before providing collateral
Severity	**Low**
Description	The delegateAllTo function allows users to delegate their clisBNB to another address. However, if the user has never set a delegate before, the call will revert. This happens because the function attempts to burn clisBNB from the previous delegate. When oldDelegate is address(0), it tries to burn tokens from the zero address, which always fails—even if the burn amount is zero. As a result, users cannot set a delegate until they have first supplied slisBNB collateral and received clisBNB.
Recommendations	Consider modifying the logic to skip the _safeBurnLp call when oldDelegatee is address(0). This will allow users to set a delegate even before providing any clisBNB.
Comments / Resolution	Acknowledged.

Issue_18	Maximum number of items in mpcWallets list isn't enforced
Severity	**Low**
Description	The code allows admins to add new MPC wallets to the mpcWallets array, and several functions iterate over this array. However, there is no enforced limit on the maximum number of wallets that can be added. This presents a potential risk: if too many wallets are added, functions that loop through the list—such as setMpcWalletCap()—may run out of gas (OOG) and revert. Critically, this could also block removeMPCWallet() and other essential management functions, making it impossible to recover from the bloated state and effectively locking the system into a non-functional configuration.
Recommendations	Consider enforcing a maximum number of wallets to make sure transactions won't revert because of OOG.

Comments / Resolution	Acknowledged.

Issue_19	Users may mint excessive amounts of unbacked clisBNB in the future
Severity	**Low**
Description	When a user supplies collateral, they are issued clisBNB tokens, which are dynamically updated based on changes in their collateral. However, upon liquidation or withdrawal of collateral, only the user's current balance of clisBNB tokens is burned: This means that if there are any minter contracts for clisBNB that allow spending of tokens issued by the SlisBNBProvider, users may be able to mint excessive amounts of unbacked clisBNB. Consider the following scenario: 1. An attacker calls supplyCollateral() and receives clisBNB tokens as LP tokens. 2. The attacker then uses a clisBNB minter contract to spend the tokens—e.g., to buy another token like TokenA. 3. Eventually, the attacker withdraws the collateral. Since they have already spent all their clisBNB, zero tokens are burned. 4. The attacker can repeat this process to mint virtually unlimited amounts of unbacked clisBNB.
Recommendations	When adding new minters for clisBNB, ensure they are restricted from burning clisBNB tokens issued by the SlisBNBProvider.
Comments / Resolution	Acknowledged.

Issue_20	Missing call to disable initializers on the constructor
Severity	**Low**
Description	The initializers are not disabled when the SlisBNBProvider is deployed. This could allow anyone to frontrun the initialization of the parameters for the proxy.
Recommendations	Consider calling _disableInitializers() on the constructor to prevent anyone frontrunning the initialization of the SlisBNBProvider.
Comments / Resolution	Resolved.

BNBProvider

The BNBProvider contract is a utility contract that enables interaction with the Moolah protocol using native BNB. Since the Moolah protocol operates on ERC-20 token standards, and native BNB is not ERC-20 compliant, this contract handles the necessary wrapping and unwrapping of BNB into WBNB and vice versa. It interacts with the WBNB token contract, the WBNB-based MoolahVault, and the Moolah, allowing users to deposit, withdraw, borrow, repay, supply collateral, and withdraw collateral using BNB directly.

Appendix: Wrapped BNB

Wrapped BNB (WBNB) is an ERC-20 token on the Binance Smart Chain (BSC) that represents BNB in a wrapped format. It uses 18 decimal places and maintains a 1:1 backing with native BNB. Users can deposit BNB into the contract to receive an equivalent amount of WBNB and can withdraw WBNB to receive BNB. This allows BNB to interact with smart contracts and decentralized applications that require ERC-20 tokens.

In the context of the BNBProvider, WBNB is used to enable users to interact with the Moolah protocol using native BNB. Since Moolah operates on ERC-20 token standards, native BNB must first be wrapped into WBNB to ensure compatibility with the protocol's smart contracts. The BNBProvider facilitates this conversion by automatically wrapping BNB into WBNB as needed, allowing seamless interaction with Moolah without requiring users to manually perform the wrapping process.

Core Invariants:

INV 1: Sender has to be authorized to withdraw or borrow from a position

INV 2: Only Moolah can call liquidate()

Privileged Functions

- _authorizeUpgrade

Issue_21	borrow() may lead to stuck assets and loss for users
Severity	**High**
Description	In the borrow() function, users can specify either assets or shares: *function borrow(* *MarketParams calldata marketParams,* *uint256 assets,* *uint256 shares,* *address onBehalf,* *address payable receiver* *) external returns (uint256 _assets, uint256 _shares) {* *...* *(_assets, _shares) = MOOLAH.borrow(marketParams, assets, shares, onBehalf, address(this));* The outputted assets from the borrow are returned in _assets. However, upon transferring the assets to the specified receiver, the following amount is transferred: *(bool success,) = receiver.call{ value: assets }("");* Due to passing assets instead of _assets as value, the receiver may actually lose their tokens, which will become stuck into the BNBProvider. This is because when the user specifies non-zero shares as input, assets will stay 0, therefore transferring 0 assets to the receiver.
Recommendations	Consider changing the value in the low-level call from assets to _assets.
Comments / Resolution	Resolved by following the recommendation.

Issue_22	Potential DoS in repay()
Severity	**Medium**

Description	In the repay() function users can specify either shares or amount to repay.
	If they specify shares, wrapAmount will be calculated based on the market's totalBorrowAssets and totalBorrowShares:
	if (wrapAmount == 0) {
	// If assets is 0, we need to wrap the shares amount
	require(shares > 0, ErrorsLib.ZERO_ASSETS);
	Market memory market = MOOLAH.market(marketParams.id());
	wrapAmount = shares.toAssetsUp(market.totalBorrowAssets, market.totalBorrowShares);
	}
	The market.totalBorrowAssets and market.totalBorrowShares are fetched directly from Moolah's storage, representing the most recently saved values. However, these variables may be stale, as interest is not accrued when the rate is previewed. As a result, wrapAmount might be less than the actual amount needed to repay the shares, causing repay() to revert due to insufficient balance when the user provides the amount in shares (assets = 0).

Recommendations	Consider calling Moolah::_accrueInterest() prior to previewing the wrapAmount.
Comments / Resolution	Resolved by following the recommendation.

Issue_23	BNBProvider does not support any of the Moolah callbacks
Severity	**Medium**
Description	The following functions support callback logic in Moolah: • repay() - triggers a callback to the msg.sender via onMoolahRepay() • supplyCollateral() - triggers a callback to msg.sender via onMoolahSupplyCollateral() However, these callbacks are no longer applicable when interacting via the BNBProvider. Additionally, users can only supply collateral through the provider due to the following require statement, which restricts direct interaction: *if (providers[marketParams.collateralToken] != address(0)) {* *require(msg.sender == providers[marketParams.collateralToken], ErrorsLib.NOT_PROVIDER);* *}* As a result, users can no longer utilize the onMoolahSupplyCollateral() callback for markets where collateralToken == WBNB, since collateral must be supplied via the BNBProvider. Similarly, the onMoolahRepay() callback becomes inaccessible when repaying through the BNBProvider (it may be used if repay() is called directly on Moolah).
Recommendations	Consider forwarding the onMoolahRepay() and onMoolahSupplyCollateral() callbacks to the sender in BNBProvider.
Comments / Resolution	Acknowledged.

Issue_24	Insufficient msg.value validation in repay()
Severity	Low
Description	The repay() function performs the following validation on the msg.value provided by the sender: *require(msg.value >= assets, "invalid BNB amount");* This check is intended to ensure that the user provides sufficient msg.value to properly execute a repay() operation. However, since repay() also allows repayment using shares, the assets parameter may be zero in such cases. As a result, the validation becomes ineffective, as it will always pass when assets == 0. This opens up a vulnerability where an underfunded repayment can still proceed, potentially utilizing excess BNB held in the BNBProvider contract to cover the shortfall.
Recommendations	Consider validating msg.value against the computed wrapAmount when the user provides shares instead of assets.
Comments / Resolution	Resolved by following the recommendation.

Issue_25	Missing call to disable initializers on the constructor
Severity	Low
Description	The initializers are not disabled when the BNBProvider is deployed. This could allow anyone to frontrun the initialization of the parameters for the proxy.
Recommendations	Consider calling _disableInitializers() on the constructor to prevent anyone frontrunning the initialization of the BNBProvider.
Comments / Resolution	Resolved by following the recommendation.