ListaDAO
Smart Collateral
+Liquidators Extension

FINAL REPORT

September ‘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	ListaDAO - Smart Collateral
Website	Lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/moolah/tree/c2d45425c23687a722cc2ef1893073f9a89a1fb8
Resolution 1	https://github.com/lista-dao/moolah/tree/b314a8757ba62e493fe918a47e9f931fadc0454e/src
Resolution 2	https://github.com/lista-dao/moolah/tree/60beac2ce6e0ad0eeccc0eec6c9ef903c9f69e0d
Resolution 3	https://github.com/lista-dao/moolah/tree/952ddae5fd8224eee919e3ee3cf71dc409a8f3e6

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	7	6		1		
Medium	4	3		1		
Low	10	9		1		
Informational	14	13			1	
Governance						
Total	35	31		3	1	

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

2. Detection

SmartProvider

The SmartProvider contract serves as the intermediary between the main Moolah contract and the new StableSwap DEX contracts. It allows LP positions from the StableSwap contracts to be used as collateral within Moolah. The SmartProvider holds the DEX's LP tokens and issues its own token, the StableSwapLPCollateral, which is what Moolah accepts as the collateral token.

The SmartProvider is the entry point for users providing StableSwap LP positions as collateral. Users do not deposit LP tokens directly, and instead, the contract accepts the two underlying tokens from users and handles entering or exiting the StableSwap pool on their behalf.

The SmartProvider also has the necessary liquidation logic required by each Moolah provider. This logic burns the StableSwapLPCollateral token received by the liquidator during liquidation, exits an equivalently sized LP position, and transfers the underlying tokens to the liquidator. The liquidator can specify minAmount0 and minAmount1 values, which represent the minimum amount of each token that must be returned to them during the liquidation.

The SmartProvider is also the oracle used by the Moolah contract for valuing the LP token collateral. The oracle logic calculates how much of each underlying token can be redeemed from one LP token, then multiplies those amounts by the oracle price of each token to determine the LP token's total value.

The SmartProvider contract is upgradeable and can be upgraded by the DEFAULT_ADMIN_ROLE.

Core Invariants:

INV 1: The balance of StableSwapLP in the SmartProvider is equal to the total supply of StableSwapLPCollateral.

INV 2: All StableSwapLP tokens within the Moolah system are stored in the SmartProvider.
INV 3: The collateralToken field always equals the StableSwapLPCollateral (TOKEN) whenever a MarketParams struct is used in a function.
INV 4: Non-zero msg.value is only allowed if one of the two tokens is BNB.
INV 5: Only the DEX contract can transfer BNB into the SmartProvider via receive.

Issue_01	redeemLpCollateral can be called with moolah as the liquidator address to redeem all LP collateral and brick funds
Severity	**High**
Description	As redeemLpCollateral is permissionless, anyone can call the function with any address as the liquidator. Specifically, calling it with the Moolah contract as the liquidator and the entire LP balance of the smart provider results in the redemption of the LP tokens and the redeemed tokens are sent to the Moolah contract, thus bricking any further collateral withdraws.
Recommendations	Remove the liquidator parameter and replace with msg.sender
Comments / Resolution	Fixed following recommendations.

Issue_02	Incorrect LP price when tokens have different decimals
Severity	**High**
Description	The peek function prices LP tokens by summing the USD value of the underlying redemption amounts as follows: *amounts[0] * price0 + amounts[1] * price1* Here, amounts represent the raw underlying token amounts returned by redeeming 1 LP token. On the other hand, price0 and price1 are 8-decimal USD oracle prices and do not depend on token decimals. Since amounts[i] are not normalized before multiplying by the prices, the above calculation is incorrect whenever the two tokens have different decimals, which results in an incorrect LP valuation.
Recommendations	Ensure that the values within the amounts array are normalized to a common decimal amount.
Comments / Resolution	Resolved with the new pricing formula.

Issue_03	Incorrect LP price when tokens are not 18 decimals
Severity	High
Description	In addition to the mismatch issue when tokens have different decimals, the peek function also miscalculates the LP token value when the two tokens share the same decimals but the decimals are not 18. The full calculation performed by peek is: *(amounts[0] * price0 + amounts[1] * price1) / 1 ether* Here, price0 and price1 are in 8 decimal places, the amounts values are in their respective token's decimals, 1 ether equals 1e18, and the intended output of peek() is an 8-decimal USD value. The issue is that dividing by 1 ether implicitly assumes the amounts array is scaled to 18 decimals, so that the division cancels out the 18 decimals and leaves only the 8 decimals from the price0/price1 values. If the tokens use any other decimal value, the result will be incorrect even if both tokens have matching decimals.
Recommendations	Ensure that the values within the amounts array are normalized to specifically 18 decimal places.
Comments / Resolution	Resolved with the new pricing formula.

Issue_04	LP valuation formula can be manipulated
Severity	High
Description	As mentioned previously, the LP valuation logic prices LP tokens by calculating how much of each underlying token would be redeemed with 1 LP token and then multiplying those amounts by the oracle price of each token. Although the token prices themselves come from a trusted oracle, the pool reserves can be influenced by anyone, so it must be considered whether it's possible to manipulate the LP price calculation by manipulating the underlying pool. In fact, this exact methodology used for valuing LP tokens has resulted in past exploits in other protocols. The core issue is that if the LP price depends on the redeemable amounts of each token, an attacker can shift the pool so that it temporarily holds more of the more expensive token. This will inflate the calculated LP price. In the context of the Moolah system, a successful manipulation of this kind would allow an attacker to deposit collateral, distort the pool to inflate the collateral's value, borrow more than should be possible, and then return the pool to its original state. The end result is that the attacker walks away with borrowed assets while the Moolah contract is left with bad debt. Note that in the StableSwap codebase, the checkPriceDiff function enforces that the reserves of the two tokens do not deviate beyond a certain threshold from the expected ratio implied by the oracle prices. This does help limit manipulation of the LP oracle price. However, the effectiveness depends on how strict the threshold is, and if it is set too loosely, manipulation would still be possible. Moreover, since safer LP oracle pricing formulas can be designed, checkPriceDiff should not be relied on as the primary line of defense for this type of exploit.
Recommendations	Consider changing the LP token valuation logic so it cannot be manipulated via the underlying pool. A common approach in DeFi

	protocols is to use the pool's get_virtual_price, multiplied by the minimum oracle price of the two tokens. Since get_virtual_price does not change with an unbalanced pool, it cannot be manipulated, and anchoring to the lower token price ensures the LP value can only be underestimated, not overestimated. The tradeoff is that this method is more conservative and reduces capital efficiency, but it is safer than relying on reserve-based pricing. Ref: Chainlink, Sky
Comments / Resolution	Resolved as per the get_virtual_price recommendation.

Issue_05	Bad timing causes liquidations to revert.
Severity	Low
Description	When a provider is configured for a market whose collateral token is an LP-collateral token, the liquidator must specify the minimum expected token amounts from the StableSwapPool during liquidation. If a large swap is executed in the pool before the liquidation transaction is included, the pool may become imbalanced (up to 3%). If this imbalance causes the received amounts to fall below the liquidator's specified minimums, the liquidation will revert, potentially leading to bad debt. **Example scenario:** 1. Assume the LP token pool used as collateral is initially balanced: - *balance[i] = 1000e18* - *balance[j] = 1000e18* - *TotalSupply = 2000e18* 2. If the liquidator aims to seize 100e18 LP tokens, they expect to receive 50e18 of each underlying token. Since remove_liquidity does not incur fees, the liquidator sets the minimum expected amounts slightly lower to account for slippage: - *minAmount[i] = 49e18* - *minAmount[j] = 49e18* 3. While the liquidation transaction is pending in the mempool, a large swap (100e18 i → 100e18 j - fee) is executed. This causes the pool to become imbalanced and leads to the liquidation reverting: - *balance[i] = 1100e18* - *balance[j] = 900e18* - LP redemption yields: - *amount[i] = 45e18* → triggers revert due to falling below minAmount[i]. - *amount[j] = 55e18*

	4. The liquidation fails because the slippage check on minAmount[i] is no longer satisfied. Even though it seems intended to revert if the liquidator incurs in more slippage than intended, that is not a sensible decision as it can cause delayed liquidations, which lead to bad debt in the affected moolah market.
Recommendations	Consider transferring the LP collateral tokens directly to the liquidator, and adding isolated functionality to redeem those LP collateral tokens in a separate transaction.
Comments / Resolution	Fixed as recommended. The liquidation flow now allows the liquidator to choose whether to redeem the collateral LP tokens during the liquidation or not. Whenever the liquidator wishes to redeem the collateral LP tokens, they can call redeemLpCollateral in the SmartProvider. Note that collateral LP tokens are still not transferable (other than by Moolah), so nothing else can be done with these tokens while they're held by the liquidator awaiting redemption.

Issue_06	Unnecessary approvals during withdrawals
Severity	**Low**
Description	In the withdrawCollateralImbalance and withdrawCollateralOneCoin functions, the SmartProvider approves its LP tokens to the DEX contract. This approval is unnecessary because the DEX functions ultimately invoked (remove_liquidity_imbalance and remove_liquidity_one_coin) burn LP tokens directly from the caller without requiring approval.
Recommendations	Remove the token approvals in the two functions.
Comments / Resolution	Fixed as per the recommendation.

Issue_07	Impossible to use data argument in supplyCollateral
Severity	**Informational**
Description	The supplyCollateral function allows the user to provide a bytes data argument, which is forwarded in the MOOLAH.supplyCollateral call. In the Moolah contract, if a non-empty data argument is provided, the onMoolahSupplyCollateral callback will be invoked on the msg.sender. Since the SmartProvider contract does not implement the onMoolahSupplyCollateral function, this will always revert when a non-empty data argument is provided by the user.
Recommendations	Consider removing the ability for the user to provide the data argument to the supplyCollateral function in the SmartProvider.
Comments / Resolution	Fixed as per the recommendation.

Issue_08	MANAGER role is unused
Severity	**Informational**
Description	The MANAGER role is defined and granted to the _manager address in the initialize function. However, this role appears to be unused and does not have any powers in the contract.
Recommendations	Consider removing the MANAGER role logic.
Comments / Resolution	Fixed as per the recommendation.

Issue_09	Redundant and imperfect maximum/minimum checks
Severity	**Informational**
Description	In the withdrawCollateralImbalance function, the maxCollateralAmount argument is intended to cap the number of LP tokens a user is willing to burn to receive the desired underlying tokens. This value is checked three times: first against the estimate returned by calc_token_amount, then passed into remove_liquidity_imbalance (which itself enforces the maximum), and finally compared again by checking the actual change in LP balance after the call. In practice, the single check within remove_liquidity_imbalance is sufficient on its own. The additional comparison against calc_token_amount is redundant and also imperfect, since the estimate it provides does not account for factors such as fees. Note that the same pattern appears elsewhere in the SmartProvider codebase: user-supplied minimum/maximum values are compared to values returned from estimate helper functions, and then also enforced during the actual conversion. Only the final enforcement is needed in these cases.
Recommendations	Consider removing the redundant estimate checks to simplify the codebase without changing the behavior. Specifically, supplyCollateral could drop its call to get_add_liquidity_mint_amount, _redeemLp could drop its call to calc_coins_amount, withdrawCollateralImbalance could drop its call to calc_token_amount, and withdrawCollateralOneCoin could drop its call to calc_withdraw_one_coin.
Comments / Resolution	Fixed as per the recommendation.

Issue_10	Missing method to deposit LP tokens.
Severity	**Informational**
Description	The protocol does not have a method for depositing on behalf of users who already hold LP tokens. As a result, users who want to provide collateral for Moolah must first remove their liquidity and then add it through the smart provider, incurring fees or missing out on yield.
Recommendations	Add functionality to the smart provider to allow users to deposit collateral on Moolah by transferring LP tokens directly from the caller.
Comments / Resolution	Fixed as per the recommendation.

Issue_11	Incorrect comments
Severity	**Informational**
Description	*// validate slippage before removing liquidity* Out of place comment because the slippage is not checked in SmartProvider, but in the StableSwap pool when maxCollateralAmount is passed into the remove_liquidity_imbalance function.
Recommendations	Consider removing the comment.
Comments / Resolution	Fixed following recommendations.

Moolah

The Moolah contract is the primary contract in the lending/borrowing system and handles all of the core accounting. Only minor changes were made to this contract within the audit scope. This part of the review was performed as a differential audit, and any issues present in the original implementation outside the modified code are not in scope. The previous audit report for this contract can be found here.

The in-scope changes consist of a new liquidate function that accepts a bytes payload parameter, which is forwarded to the provider during liquidation. The purpose of this addition is to enable liquidators interacting with the SmartProvider to specify a minimum amount of tokens that must be returned from the liquidation.

Issue_12	Liquidations stop in turbulent market conditions
Severity	**High**
Description	In turbulent market conditions, the price in the pool can drift from the oracle price being reported for the constituent tokens. This can happen due to a number of reasons: 1. TWAP used in resilient oracle is a lagging indicator of price, and gets outdated quickly. 2. If bridge transactions slow down due to high traffic, prices can diverge on different chains. Oracles will give price for USDT on eth, not on BNB. 3. Large liquidations can momentarily change dex prices drastically. Borrowers can protect themselves from such fluctuations by keeping a low LTV on their positions. The system protects itself by offloading the risk to liquidators, who pay off the debt in the system in return for collateral tokens. The main issue is that during liquidation, the checkPriceDiff function in the stableswap contract is called, which can revert if the pool price does not track the oracle price. This will lead to failed liquidations during volatile times, which is when the liquidations are most crucial. In case of a depeg event due to insolvency or liquidity crunch (SVB bank insolvency, stEth depeg due to withdrawal queues etc), the oracle price can quickly go out of range and block any liquidation attempts.
Recommendations	Consider removing any such checks from the liquidation flow, since it is important for those transactions to go through at any token price. Consider switching to a manipulation-resistant pricing algorithm as described in a different issue in the report, to do away with the need for the checkPriceDiff function. If the checkPriceDiff is still required, consider allowing liquidations to directly credit the caller with the collateral tokens, which can

	later be redeemed for the LP tokens. While they can still run into the same issue when redeeming, it will at least allow all liquidations to still go through. Furthermore, additional measures need to be put in place to prevent false price calculations in low/no liquidity conditions.
Comments / Resolution	Fixed by allowing liquidations and redemptions to happen separately. The liquidator receives the collateral LP tokens and can redeem them at any point via redeemLpCollateral, including after the liquidation.

Issue_13	Liquidators can choose between different liquidate functions
Severity	**Medium**
Description	The new liquidate function allows the liquidator to include an optional bytes payload parameter. If the payload is empty, Moolah forwards the liquidation call to the provider using the old function signature liquidate(bytes32,address). If the payload is non-empty, Moolah instead uses a new liquidation function with signature liquidate(bytes32,address,uint256,bytes). Because the liquidator can freely choose whether to provide an empty or non-empty payload, they can cause Moolah to call using the new signature even on old providers that only support the old liquidate function. In most cases, this will simply revert. However, if an older provider contract happens to implement a fallback function, the call would succeed and execute the fallback instead of any liquidation logic, which could have unintended consequences. Currently, it does not appear that any of the existing provider contracts have a fallback function that would be exploitable in this way, so this is only a theoretical risk.
Recommendations	Consider removing the dependency on the liquidator's input when deciding which liquidate function signature to use. A more robust approach would be to base this decision on a provider configuration in storage.
Comments / Resolution	Fixed as a consequence of the new liquidator logic. The code has returned to only having one liquidate function.

Issue_14	Incorrect verifying-contract address in the domain separator
Severity	**Informational**
Description	The domain separator is set in the constructor and is an immutable variable. This means that once set, any proxy inheriting this contract will use the same domain separator value, which contains the address of the implementation contract as the verifying contract. *DOMAIN_SEPARATOR = keccak256(abi.encode(DOMAIN_TYPEHASH, block.chainid, address(this)));* While there are no security ramifications, this goes against EIP 712, which requires the contract that consumes the signature to be specified in the domain separator. In this case, the implementation address is set in the domain separator when it should have been the actual proxy address.
Recommendations	Consider adding an update function allowing an update of the domain separator to the proxy's own address.
Comments / Resolution	Resolved by removing the immutable DOMAIN_SEPARATOR and adding a _buildDomainSeparator() function.

Issue_15	Redundant zero provider address check in addProvider
Severity	**Informational**

Description	*address token = IProvider(provider).TOKEN();*
	require(provider != address(0)
	The non-null sanity check on provider is performed after the call to the provider to fetch token, making it redundant, since it should have reverted prior.

Recommendations	Remove or shift the check before attempting to fetch token*.*
Comments / Resolution	Not fixed.

Issue_16	flashloan combined with StableSwapLPCollateral is dangerous
Severity	**Informational**
Description	With the new redeemLpCollateral in SmartProvider, users can directly redeem StableSwapLPCollateral. Combined with the Moolah flashloan function, an attacker could temporarily pull LP tokens out of the system and act on the pool's state while those tokens are out (e.g. they could swap/re-LP into a favorable position). One specific concern with this is if the Moolah contract holds the entire LP supply, and an attacker redeems them all to "reset" accrued yield back to 1:1. Technically, this attack is prevented by the checkPriceDiff guard, which makes full LP withdrawals impossible. However this may be the only thing preventing this issue. An attacker can follow these steps to possibly economically exploit the protocol: 1. Take a flashloan of stableSwapLpCollateral tokens 2. redeem the stableSwapCollateral tokens to completely mpty the pool. 3. Manipulate the pool state while at 0 liquidity/profit from fees etc. 4. Mint back LP tokens by providing liquidity. 5. Supply the lp tokens back to Moolah by calling supplyDexLp in the smartProvider. Smart provider will mint LP collateral tokens. 6. Take a loan and liquidate this new account. This might be possible since the pool was in a manipulatable state before. This will send the LPCollateral tokens to the liquidator. 7. Pay back the flashloan. While a definitive exploit has not yet been found, the attack path shows a lot of promise and should thus be addressed.
Recommendations	Since this issue is close to exploitable, we suggest explicitly preventing this attack path by adding a blacklist of tokens that cannot be used in flashloan, and blacklist all LP collateral tokens as soon as they're created.
Comments / Resolution	Fixed by adding a blacklist for flash-loanable tokens. LP tokens should be blacklisted from being flash-loaned once deployed.

StableSwapFactory

The StableSwapFactory is the main factory contract responsible for creating new StableSwap token pairs. For each pair, the factory deploys two contracts: a StableSwapPool and a StableSwapLP. Both are deployed as ERC1967Proxy instances, and the implementation addresses they initially point to are configurable values.

The factory itself will also be deployed behind a proxy and is upgradeable by the DEFAULT_ADMIN_ROLE.

The StableSwapFactory implementation is forked from the PancakeStableSwapFactory contract from PancakeSwap.

Core Invariants:

INV 1: Pools created by the factory are automatically set as the minter of their LP tokens.

INV 2: The factory retains no ownership or privileged roles in the contracts it deploys once the deployment process is finished.

Privileged Functions

- createSwapLP
- createSwapPair
- addPairInfo
- setImpls

Issue_17	Wrong initialization of DEPLOYER role identifier in proxy setup
Severity	**Medium**
Description	The contract defines the DEPLOYER role identifier as: *bytes32 public DEPLOYER = keccak256("DEPLOYER");* Since DEPLOYER is not declared as constant or immutable, Solidity treats it as a state variable. This means the computed hash is stored in the implementation contract's storage at deployment time. When the contract is used through a proxy (UUPS), the proxy relies on its own storage layout. Because the proxy's storage slot for DEPLOYER is never initialized, the value resolves to 0x00...00. This creates an overlap: OpenZeppelin's DEFAULT_ADMIN_ROLE is also defined as 0x00...00. As a result, checks intended for the DEFAULT_ADMIN_ROLE role instead match the deployer role. This effectively grants deployers or other accounts unintended admin-level privileges, such as calling addPairInfo, setImpls, or upgrading the contract itself.
Recommendations	Declare DEPLOYER as a constant so that the value is embedded in the bytecode rather than stored in contract storage: *bytes32 public constant DEPLOYER = keccak256("DEPLOYER");*
Comments / Resolution	Fixed as per the recommendation.

Issue_18	Data overwriting on stableSwapPairInfo mapping
Severity	Low
Description	The nested mapping stableSwapPairInfo is set in the addPairInfoInternal function, which is invoked by both addPairInfo and createSwapPair. Currently, there is no check to prevent overwriting existing entries when deploying multiple pools with the same token pair. Deployer intent may include launching multiple pools with identical tokens but differing amplification factors or fee rates. These parameters influence the risk profile and appeal to distinct user groups. When multiple pools are deployed with the same token pair, the existing entry in the stableSwapPairInfo mapping is overwritten. As a result, calls to getPairInfo may return incorrect data, referencing the most recently deployed pool rather than the intended one.
Recommendations	Update the stableSwapPairInfo mapping or introduce a new data structure capable of supporting multiple pools with the same token pair but different configurations.
Comments / Resolution	Resolved by changing stableSwapPairInfo into an array that can hold multiple entries.

Issue_19	createSwapLP does not need to be public
Severity	Informational
Description	The createSwapLP function is a helper function used inside createSwapPair to deploy the LP token contract. It is also exposed publicly and can be called on its own. Calling createSwapLP directly does not appear to be useful, since there is no mechanism within the factory to link an LP token created independently with a pool deployed through createSwapPair.
Recommendations	Consider changing createSwapLP to be an internal function.
Comments / Resolution	Resolved as per the recommendation.

Issue_20	Redundant sorting in _createSwapPair function
Severity	Informational
Description	The internal _createSwapPair function sorts the supplied tokenA and tokenB tokens. This is unnecessary, since the parent createSwapPair function already sorts the tokens and calls the internal function with the sorted result.
Recommendations	Consider removing the sorting in the internal function, only if the _createSwapPair function is not going to be called from any other function in the future.
Comments / Resolution	Resolved as per the recommendation.

StableSwap Pool

The StableSwapPool is the core DEX contract that facilitates swaps and LP actions. The pool's math follows Curve's StableSwap invariant, and the Solidity implementation is forked from PancakeSwap's PancakeStableSwapTwoPool.

The code introduces some smaller changes from the source code: the contract is now upgradeable, admin functions use AccessControlEnumerableUpgradeable roles instead of an explicit owner variable, and pausing functionality no longer uses an is_killed storage flag and instead uses PausableUpgradeable.

In addition to these smaller changes, there is one major change: at the end of every function that performs swaps or modifies LP positions, the contract calls a new checkPriceDiff function. The goal of this function is to compare the ratio of the oracle prices of the two tokens to the ratio of their reserves in the pool. If the difference exceeds some configurable thresholds, the call will revert.

Core Invariants:

INV 1: After every state-changing action that moves pool balances, checkPriceDiff is executed and must succeed.

INV 2: Within checkPriceDiff, the exchange rates implied by get_dy are within price0DiffThreshold/price1DiffThreshold of the oracle prices.

Privileged Functions

- ramp_A
- stop_rampget_A
- commit_new_fee
- apply_new_fee
- revert_new_parameters
- withdraw_admin_fees
- donate_admin_fees
- changePriceDiffThreshold
- pause
- unpause

Issue_21	Tokens with large prices can lead to checkPriceDiff reverting even when balanced
Severity	**High**
Description	As part of its calculations, checkPriceDiff computes dy0 and dy1, which represent the output token amounts if one full unit of each input token (i.e. 10**token_decimals) were swapped in the pool. These values are then used to check whether the pool is balanced based on the oracle prices. This approach can create issues for tokens with high token values, since swapping an entire unit may represent an extremely large trade relative to pool size. For example, consider if the pool consists of two Bitcoin-related tokens. One full Bitcoin is currently worth over $100k. The dy0 and dy1 calculations would be based on a swap of one whole Bitcoin, which could incur significant price impact even if the pool is otherwise balanced (e.g. if the pool has 0.1 = $10k of each token in reserves). Since balanced pools can still lead to checkPriceDiff() reverts, there may be scenarios where LP positions cannot be fully withdrawn, and liquidations unexpectedly revert.
Recommendations	Change the checkPriceDiff logic to compute dy0 and dy1 with a fixed dollar amount swap, instead of a full swap of each token.
Comments / Resolution	Fixed following recommendations. The simulated swap logic uses a $100 swap for each token now. Note that it may be preferable to lower this dollar value even more.

Issue_22	checkPriceDiff is prone to failures in low/zero liquidity conditions
Severity	**High**
Description	The checkPriceDiff function is used to check the price of the pool tokens after an interaction with the pool. The issue is that due to the way the pool price is calculated, there can be issues when there is very little liquidity in the pool. *uint256 dy1 = get_dy(0, 1, dx0);* The main step when calculating the price is calculating the exchange rate with a simulated swap. This is done by swapping a unit amount of token0 for token1 or vice versa. The issue is that if there is only a few wei of liquidity left, this can give extremely off prices. Furthermore, if there is no liquidity left in the pool at all, this can even revert due to division by zero. Such a situation can happen if a user tries to remove the entire liquidity from the pool. The remove_liquidity function first removes the liquidity from the pool and then does the price check. So during the price check step, there can be no liquidity left in the pool, which can lead to reverts. This is a serious issue since it is encountered during the liquidation in the Moolah contract. The liquidate function in the smart provider removes liquidity from the stableswap pool and sends it to the liquidator. However, if this leads to the pool being left with little to no funds, the liquidation function itself will revert.
Recommendations	Consider switching to a manipulation-resistant pricing algorithm as described in a different issue in the report, to do away with the need for the checkPriceDiff function. Any revert based on market/liquidity conditions(ex checkPriceDiff) must not be encountered in the liquidation flow. If the checkPriceDiff is still required, consider allowing liquidations to directly credit the caller with the collateral tokens, which can later be redeemed for the LP tokens. While they can still run into the same issue when redeeming, it will at least allow all liquidations to still go through. Furthermore, additional measures need to be put in place to prevent false price calculations in low/no liquidity conditions.

Comments / Resolution	Acknowledged, checkPriceDiff is still present in the code. Although the liquidation flow has been altered with new liquidation contracts, the problems noted in this issue still exist, for example withdrawing all liquidity from the pool is impossible. The client has decided to acknowledge this.

Issue_23	All pool operations freeze during volatile conditions.
Severity	**Medium**
Description	Due to the presence of the checkPriceDiff function in all pool interaction functions, all pool functions will stop if the state of the pool does not match the oracle data, within a range. While this is unlikely in stableswap pools, this can still happen in case of token depegs or bridge failures. In such scenarios, swaps, liquidity additions, and liquidity removals all stop, only allowing swaps that change the price back to the oracle price, which can be heavily outdated by this point. Thus, the pool can become unusable for long periods of time, since it essentially locks down, only allowing swaps which push the pool price to the oracle-reported value, which can be an unprofitable condition for arbitrage bots. Arbitrage bots are essential for price discovery. Any mechanism that prevents such bots from operating, or forces operations only on certain quantities of tokens, risks leaving the pool with a price different from the wider ecosystem.
Recommendations	Consider removing the checkPriceDiff function from the pool operations and instead using a manipulation-resistant pricing algorithm to value the LP tokens.
Comments / Resolution	Acknowledged., checkPriceDiff is still present in the code. Although the LP token pricing has a new formula, the problems noted in this issue still exist. The client has decided to acknowledge this.

Issue_24	checkPriceDiff logic includes fees
Severity	Medium
Description	The checkPriceDiff function relies on get_dy to simulate a swap of one full token in each direction. The returned output amounts are then combined with oracle prices to compute an implied price from the pool, which is compared back against the other token’s oracle price and reverts if the deviation is too large. However, note that the get_dy function returns the swap amount after fees. This means that even in a perfectly balanced pool, the price will deviate from the oracle price simply due to fees. Due to this issue, pools will revert under normal conditions, not because of real price divergence, but simply because the fee is baked into the measurement, turning the guard into a function of the fee configuration rather than market reality.
Recommendations	If checkPriceDiff is used in the future, consider updating the logic to ignore fees when deriving the pool’s implied token prices.
Comments / Resolution	Fixed as per the recommendation.

Issue_25	Precision loss on get_dy and get_dy_underlying
Severity	Low
Description	The core logic of the get_dy and get_dy_underlying functions follows this sequence: 1. Scale dx and calculate new x 2. Calculate new y using get_y 3. Calculate dy and convert into token decimals 4. Calculate fee In contrast, the actual exchange function executes operations in the following order: 1. Scale dx and calculate new x 2. Calculate new y using get_y 3. Calculate dy and fee 4. Convert amounts into token decimals By converting to token decimals before calculating the fee, the get_dy and get_dy_underlying functions introduce minor precision loss. Specifically, the fee will be under-calculated by 1 wei, leading to slight discrepancies compared to the actual swap output, which breaks expectations.
Recommendations	Consider acknowledging this issue since it's a single wei fee difference and prevents unnecessary changes from a well-established codebase. However, this means that the result of get_dy cannot be directly used as the minimum output for the exchange function, since that might revert. Otherwise, the execution flow in get_dy and get_dy_underlying should be aligned with the actual swap function to ensure consistent and accurate results.
Comments / Resolution	Acknowledged.

Issue_26	Wrong initialization of bnb_gas due to proxy setup
Severity	Low

Description	The contract initializes the variable bnb_gas during deployment as follows:
	uint256 public bnb_gas = 4029;
	This sets the value in the implementation contract's storage. However, when the contract is used via a proxy, that storage slot remains uninitialized in the proxy's context. As a result, the bnb_gas value defaults to zero unless explicitly set.
	This affects BNB transfers made by the contract, as shown below:
	function _safeTransferBNB(address to, uint256 value) internal {
	(bool success,) = to.call{ gas: bnb_gas, value: value }("");
	require(success, "BNB transfer failed");
	}
	This will lead to 0 gas being forwarded for BNB transfers. This requires the admin to call set_bnb_gas for each pool in order to configure it correctly, which can be tedious and requires admin input for each pool.

Recommendations	Initialize bnb_gas within the initialize function to ensure correct storage when using the proxy.
Comments / Resolution	Fixed as per the recommendation.

Issue_27	Missing A_PRECISION for better granularity when ramping A factor
Severity	**Low**
Description	In newer Curve stable pools, a constant called A_PRECISION was added to scale the amplification factor (A). Without this scaling, when the admin ramps A up or down over time, the value only changes in whole integer steps. This creates small jumps in the effective amplification, which makes the invariant progression slightly discontinuous. While this does not break functionality, it can introduce small inconsistencies in pricing during the ramp. By adding A_PRECISION, the ramping is done with higher granularity, preventing any significant precision loss in A. This makes the curve evolve smoothly and avoids minor rounding artifacts. Link to curve's implementation → https://github.com/curvefi/curve-contract/blob/master/contracts/pool-templates/base/SwapTemplateBase.vy#L105
Recommendations	Consider introducing A_PRECISION when handling the amplification factor so that ramps are smoother and less quantized, improving accuracy and consistency.
Comments / Resolution	Fixed following recommendation.

Issue_28	Incorrect MAX_A check in ramp_A()
Severity	**Low**
Description	In the ramp_A() function, the _future_A value is compared against MAX_A after scaling by the A_PRECISION value. This is incorrect, because MAX_A represents the maximum A value before scaling. Since A_PRECISION == 100 and MAX_A == 1e6, this means that ramp_A() is incorrectly disallowing _future_A values between 10,000 and 999,999, which should actually be allowed.
Recommendations	Shift the referenced check to before _future_A = _future_A * A_PRECISION
Comments / Resolution	Fixed following recommendations.

Issue_29	Read-only reentrancy in get_virtual_price
Severity	**Low**
Description	get_virtual_price can be re-entered when removing liquidity involving BNB. There have been past exploits [Ref: https://www.zellic.io/blog/how-not-to-create-a-cdp-or-lending-protocol/#curves-get_virtual_price] Re-entering with the current Stableswap pool implementation should not be possible because _safeTransferBNB limits the amount of gas sent with BNB (default value of 4029 gas), but it would be good to have to mitigate any behavioral changes.
Recommendations	Use the internal _reentrancyGuardEntered method & check either in get_virtual_price or prior to calling it in the smart provider (which then requires exposing the internal checker to external)
Comments / Resolution	Fixed following recommendations.

Issue_30	TokenExchange event fee values have different normalization
Severity	**Informational**
Description	The TokenExchange event has a dy_fee parameter and a dy_admin_fee parameter. In the code leading up to the event being emitted, the dy_admin_fee is un-normalized by multiplying by PRECISION and dividing by RATES[j]. On the other hand, the dy_fee is still a normalized value when it is emitted in the event. This means whenever the token does not have 18 decimal places, the two parameters will have different units, which may be unexpected.
Recommendations	Consider un-normalizing the dy_fee before it is emitted in the TokenExchange event by multiplying by PRECISION and dividing by RATES[j].
Comments / Resolution	Fixed as per the recommendation.

Issue_31	Redundant coins array length check
Severity	**Informational**
Description	The initialize function implements a check that validates that the length of the coins array provided matches N_COINS. However, this check is already enforced since the input parameter for the initialize function already specifies the array length expected, and is thus redundant. *function initialize(* *address[N_COINS] memory _coins,* *//...*
Recommendations	Consider removing the coins array length check.
Comments / Resolution	Fixed as per the recommendation.

Issue_32	Missing update functionality for the oracle
Severity	**Informational**
Description	If the oracle of StableSwapPool fails for any reason, most of the protocol functionality will fail. As there is no way to update the oracle it could remain block until some upgrade.
Recommendations	Add a function to update the oracle address in case it fails.
Comments / Resolution	Fixed following recommendations.

Issue_33	Incorrect comments
Severity	**Informational**

Description	The comments have not been updated to reflect that only $100 of tokens are being swapped.
	uint256 dy1 = get_dy_without_fee(0, 1, dx0); // token1Amount for 1 token0, in original precision
	uint256 dy0 = get_dy_without_fee(1, 0, dx1); // token0Amount for 1 token1, in original precision

Recommendations	Update the comments.
Comments / Resolution	Fixed following recommendations.

StableSwapPoolInfo

The StableSwapPoolInfo contract is a helper contract for performing various calculations related to StableSwap pairs. It is forked from PancakeSwap's PancakeStableSwapTwoPoolInfo, with only minor differences: the contract is upgradeable by the DEFAULT_ADMIN_ROLE, and additional return values have been added to get_add_liquidity_fee() and get_remove_liquidity_one_coin_fee().

Within the audit scope, its primary use is within the SmartProvider contract, which uses its get_add_liquidity_mint_amount() and calc_coins_amount() functions to convert between LP token amounts and underlying token amounts.

Issue_34	Function calc_coins_amount reverts when totalSupply is zero
Severity	**Low**
Description	The calc_coins_amount function is called by the peek function in SmartProvider, which calculates the price of each LP token. If the total supply of LP tokens is zero, the following operation causes a division by zero and results in a revert: *uint256 value = (IStableSwap(_swap).balances(i) * _amount) / total_supply;* This, in turn, causes the peek function to revert, potentially blocking flows that rely on this pricing logic. While we have not identified a practical exploit path, it is advisable to address this issue for improved robustness. Similarly, the get_virtual_price function in the stableswap contract also reverts in case of no liquidity being present.
Recommendations	Update the calc_coins_amount function to return zero early when totalSupply is zero. Additionally, ensure that the pool contains sufficient liquidity before enabling its LP tokens as collateral within Moolah.
Comments / Resolution	Resolved as per the recommendation.

StableSwapLP

The StableSwapLP token represents LP positions in a specific StableSwap pool. The contract inherits from OpenZeppelin's ERC20Upgradeable, with the only additions being the mint and burnFrom functions. Each pool has its own StableSwapLP contract and has the ability to mint and burn tokens.

This contract is upgradeable by the DEFAULT_ADMIN_ROLE.

Core Invariants:

INV 1: The minter address of the StableSwapLP is the corresponding StableSwap pool.

Privileged Functions

- setMinter
- mint
- burnFrom

No issues were identified in this contract.

StableSwapLPCollateral

The StableSwapLPCollateral is the collateral token used for StableSwap LP positions within the Moolah system. The SmartProvider contract holds the StableSwapLP tokens minted by the pool and mints/burns an equivalent amount of StableSwapLPCollateral to be used inside Moolah.

The contract inherits OpenZeppelin's standard ERC20Upgradeable contract. The transfer function is overridden so that only the Moolah contract can call it, and it also includes mint and burn functions to be used by the SmartProvider. The contract is upgradeable by the DEFAULT_ADMIN_ROLE.

Note that this contract is not deployed by the StableSwapFactory. It must be deployed separately and configured with the SmartProvider contract before it can be used in the Moolah system.

Core Invariants:

INV 1: The minter address of the StableSwapLPCollateral is the corresponding SmartProvider.

Privileged Functions

- setMinter
- mint
- burn
- transfer

Issue_35	Partial access check on token transfers
Severity	**Low**
Description	The transfer function is overridden with an onlyMoolah modifier, so only the Moolah contract is allowed to transfer the collateral tokens. However, the contract does not apply the same modifier to the transferFrom function, so the tokens are actually still transferable.
Recommendations	Consider removing the onlyMoolah modifier, since users cannot get a hold of the collateral tokens outside of flashloans, and even then, those tokens aren't redeemable for underlying assets. If the modifier is still necessary, consider adding the same modifier to the transferFrom function as well.
Comments / Resolution	Fixed as per the recommendation.

ListaDAO - Liquidators Extension

Project	ListaDAO - Liquidators Extension
Website	Lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/moolah/tree/60beac2ce6e0ad0eeccc0eec6c9ef903c9f69e0d
Resolution 1	https://github.com/lista-dao/moolah/commits/41ba61b7d2d84cde4d6fd7293a2ba425c76936d4/
Resolution 2	https://github.com/lista-dao/moolah/tree/565ce42ff7589ed70658c38a91491c6f2c5a559e

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	3	3				
Medium	6	6				
Low	2	2				
Informational	10	6	1	3		
Governance						
Total	21	17	1	3		

Detection

Liquidator

The Liquidator contract is built on top of the core Moolah codebase. The main purpose of the liquidator is to wrap liquidate calls to the core Moolah contract. For certain markets, the liquidator contract may be a privileged address within the whitelisted set of addresses authorized to perform liquidations for that market. All functions in this contract are restricted to the MANAGER role, which configures parameters, or the BOT role, which executes the liquidations and swaps.

This part of the review was performed as a differential audit, and any issues present in the original implementation outside the modified code are out of scope.

The main in-scope changes consist of two new liquidation functions that support liquidating smart collateral positions. liquidateSmartCollateral uses the contract's existing loanToken balance to perform the liquidation. After the liquidation, it calls redeemLpCollateral to convert the StableSwapLPCollateral into its two underlying tokens. flashLiquidateSmartCollateral differs in that it redeems the LP tokens during the onMoolahLiquidate callback and can optionally swap the underlying tokens into the loanToken as needed for the liquidation. Note that it is also possible to use the existing liquidation functions on smart collateral positions, which would avoid the automatic redeemLpCollateral call.

Another in-scope addition is the sellBNB function, which allows the BOT to initiate a swap to sell the contract's BNB into another token.

The remaining in-scope changes are minor and simply update onMoolahLiquidate to branch between the old and new logic.

Core Invariants:

INV 1: swapCollateral and swapSmartCollateral cannot both be true

INV 2: liquidateSmartCollateral and flashLiquidateSmartCollateral must be called on markets where the collateral matches the smart provider's TOKEN return value

INV 3: For flashLiquidateSmartCollateral, LP redemption alone must cover the required loanToken amount.

INV 4: Liquidations are only allowed on whitelisted market ids

INV 5: token0Pair and token1Pair call targets must be whitelisted

INV 6: All token approvals granted during onMoolahLiquidate are revoked at the end of the function (except for the loanToken approval to Moolah)

Privileged Functions

- sellBNB
- liquidateSmartCollateral
- flashLiquidateSmartCollateral

Issue_01	Liquidations are still dependent on pool withdrawals.
Severity	High
Description	Both contracts, Liquidator and PublicLiquidator, enforce a liquidation of smart collateral to be followed by a liquidity withdrawal from the pool. This decision can cause a DoS on liquidations when the market prices are volatile due to a revert on the liquidity withdrawal from the pool. This vulnerability has previously been identified and described in the issues named "*Liquidations stop in turbulent market conditions*" and "*Bad timing causes liquidations to revert*". While the issue was solved on the SmartProvider contract, it still persists in the liquidator contracts, as these still enforce a liquidity withdrawal just after the liquidation.
Recommendations	Modify the logic in Liquidator and PublicLiquidator to allow the caller to choose whether to avoid calling redeemLpCollateral just after a smart collateral liquidation.
Comments / Resolution	Fixed by adding a redeemSmartCollateral function to both liquidator contracts, and adding the liquidateWithCollTransferOpt function to the PublicLiquidator. Liquidations can be done using the standard liquidation logic, and the LP smart collateral tokens remain in the contracts to be redeemed separately in a follow-up step.

Issue_02	Incorrect balance accounting when underlying token equals loanToken
Severity	**Medium**
Description	In onMoolahLiquidate, the swapSmartCollateral logic measures the loanToken balance before and after swapping the two underlying tokens (from the LP redemption) into the loanToken. The increase in the loanToken balance must be at least repaidAssets, so the swap of the redeemed tokens is expected to produce enough loanToken to cover the liquidation. There is also now logic to skip the swap for any underlying token that is already the loanToken. However this logic is currently flawed when one of the underlying tokens is the loanToken. The "before" loanToken balance is taken after the LP tokens are redeemed, so any loanToken obtained directly from the redemption is already included in that balance and cannot contribute to the measured increase. The swap is skipped for that token and only the swap of the other underlying token can increase the loanToken balance from there. The increase due to the other token alone is likely not enough to meet the repaidAssets threshold, so the liquidation would fail even though enough loanToken was obtained from the redemption and swaps.
Recommendations	Take the before balance of loanToken prior to doing the LP redemption.
Comments / Resolution	Fixed as recommended.

Issue_03	smartProvider argument could be malicious
Severity	Medium
Description	The liquidateSmartCollateral and flashLiquidateSmartCollateral functions accept a smartProvider address as input. The code verifies that ISmartProvider(smartProvider).TOKEN() == params.collateralToken. The smartProvider is also used to call redeemLpCollateral, and depending on the flow, token(0), token(1), or dexLP. It's possible for the provided address to not actually be a real smartProvider, but instead a malicious contract that returns values crafted to pass the checks. Since the liquidator can operate across many markets, a malicious contract could return a real collateral token (that is not related to a smart provider) when TOKEN is called. This can be used to steal any tokens present in the PublicLiquidator contract itself, by returning fabricated values of amount0 and amount1 from the redeemLpCollateral call. This will then swap such tokens already in the contract to pay the loan. This issue is valid for both the liquidator contracts.
Recommendations	Consider adding a whitelist of smart providers an checking the passed value against that.
Comments / Resolution	Fixed as recommended.

Issue_04	Incorrect event emission on SellToken.
Severity	**Informational**
Description	The last field of the SellToken event is named amountOutMin. *event SellToken(address pair, address tokenIn, address tokenOut, uint256 amountIn, uint256 amountOutMin);* However, when the event is emitted in sellToken and sellBNB, the actual value emitted in the last field is actualAmountOut. *emit SellToken(pair, BNB_ADDRESS, tokenOut, amountIn, actualAmountOut);*
Recommendations	Consider either: • Updating the last field name in SellToken event to be named actualAmoutOut. • Emit the amountOutMin value instead of the actual amount out.
Comments / Resolution	Partially resolved. The field name in the event has been changed but it has a typo ("Amout" instead of "Amount").

Issue_05	Missing natspec comment on liquidate.
Severity	**Informational**
Description	The liquidate function has 4 arguments, but the Natspec only describes 3 of them. The last argument, repaidShares, is not described in the Natspec.
Recommendations	Fix the Natspec in the function liquidate.
Comments / Resolution	Fixed as recommended.

Issue_06	Redundant parameters emitted in SmartLiquidation
Severity	**Informational**
Description	The payload is decoded to the minimum amounts, but because swapSmartCollateral is disabled, these parameters are not used. Regardless, they are emitted in the SmartLiquidation event.
Recommendations	Remove the minAmount0 and minAmount1 parameters
Comments / Resolution	Acknowledged.

Issue_07	liquidate is unnecessarily payable
Severity	**Informational**
Description	liquidate is payable, but doesn't utilise msg.value anywhere (perhaps in onMoolahLiquidate)
Recommendations	Consider if liquidate needs to be payable, remove if not needed
Comments / Resolution	Fixed as recommended.

Issue_08	Add SafeTransferLib using directive for address type for better readability
Severity	**Informational**

Description	*using SafeTransferLib for address;*
	allows for better code readability wherever the library is used.
	For instance,
	SafeTransferLib.balanceOf(params.loanToken, address(this));
	becomes
	params.loanToken.balanceOf(address(this));

Recommendations	Add the using directive.
Comments / Resolution	Fixed as recommended.

PublicLiquidator

The PublicLiquidator contract is similar to the Liquidator contract, but it allows for any address to call the main liquidation functions.

This part of the review was performed as a differential audit, and any issues present in the original implementation outside the modified code are out of scope.

The main in-scope changes consist of two new liquidation functions that support liquidating smart collateral positions. liquidateSmartCollateral transfers loanToken from the caller to perform the liquidation. After the liquidation, it calls redeemLpCollateral to convert the StableSwapLPCollateral into its two underlying tokens and sends them to the caller. flashLiquidateSmartCollateral differs in that it redeems the LP tokens during the onMoolahLiquidate callback and can optionally swap the underlying tokens into the loanToken as needed for the liquidation.

Core Invariants:

INV 1: All liquidations must pass the isLiquidatable(id, borrower) check

INV 2: For flashLiquidateSmartCollateral, LP redemption alone must cover the required loanToken amount.

INV 3: All token approvals granted during onMoolahLiquidate are revoked at the end of the function (except for the loanToken approval to Moolah)

Privileged Functions

- flashLiquidateSmartCollateral
- liquidateSmartCollateral

Issue_09	redeemSmartCollateral doesn’t return constituent tokens
Severity	High
Description	The redeemSmartCollateral function redeems the collateral tokens to withdraw liquidity from the pool. But the issue is that these tokens stay with the liquidator contract instead of being sent to the actual liquidator. This is because the liquidator contract calls redeemLpCollateral on the smart provider which sends the constituent tokens to ITS msg.sender, i.e., the liquidator contract itself. It should instead send these tokens to the actual caller.
Recommendations	Send forward the tokens from the liquidator to the caller.
Comments / Resolution	Fixed as recommended.

Issue_10	Incorrect balance accounting when underlying token equals loanToken
Severity	**Medium**
Description	In onMoolahLiquidate, the swapSmartCollateral logic measures the loanToken balance before and after swapping the two underlying tokens (from the LP redemption) into the loanToken. The increase in the loanToken balance must be at least repaidAssets, so the swap of the redeemed tokens is expected to produce enough loanToken to cover the liquidation. There is also now logic to skip the swap for any underlying token that is already the loanToken. However this logic is currently flawed when one of the underlying tokens is the loanToken. The "before" loanToken balance is taken after the LP tokens are redeemed, so any loanToken obtained directly from the redemption is already included in that balance and cannot contribute to the measured increase. The swap is skipped for that token and only the swap of the other underlying token can increase the loanToken balance from there. The increase due to the other token alone is likely not enough to meet the repaidAssets threshold, so the liquidation would fail even though enough loanToken was obtained from the redemption and swaps.
Recommendations	Take the before balance of loanToken prior to doing the LP redemption.
Comments / Resolution	Fixed as recommended.

Issue_11	Failing swaps when token0 or token1 is native BNB.
Severity	**Medium**

Description	When flashLiquidateSmartCollateral is called, the callback (onMoolahLiquidate) redeems the liquidated LP tokens and attempts to swap the underlying assets to the loan token for repayment.
	However, if one of the underlying tokens is native BNB, the entire redeemed amount is sent as msg.value:
	uint256 _value = token0 == BNB_ADDRESS ? amount0 : 0;
	(bool success,) = arb.token0Pair.call{ value: _value }(arb.swapToken0Data);
	require(success, SwapFailed());
	This logic is flawed because the redeemed amount of BNB will never exactly match the exact amount required by the swap. As a result, the swap will typically fail, since the 1Inch router enforces strict value matching on the swap function:
	if (msg.value != (srcETH ? desc.amount : 0)) revert RouterErrors.InvalidMsgValue();
	This causes all calls to flashLiquidateSmartCollateral involving BNB as one of the underlying tokens to revert. This issue is present in both Liquidator and PublicLiquidator contracts.

Recommendations	Send only the exact BNB amount required for the swap, as specified by the liquidator, rather than the entire redeemed amount.
Comments / Resolution	Resolved by using the minToken0Amt/minToken1Amt liquidator values as the native amount of token sent in the swap.

Issue_12	onMoolahLiquidate arbitrary calls can drain approvals
Severity	High
Description	When the flashLiquidateSmartCollateral function is used, the onMoolahLiquidate callback performs two additional calls after redeeming the LP tokens. These calls can be made to arbitrary addresses with arbitrary calldata provided by the user. The intent is for these calls to use the redeemed tokens and swap them into the loanToken required for the liquidation. A safety check ensures that the loanToken balance increases sufficiently to cover the liquidation amount. These arbitrary calls can be very dangerous, especially if they target token contracts. There is at least one concrete exploit path: by setting the call target to the loanToken contract and the calldata to transferFrom(victim, publicLiquidator, victim's balance), an attacker could transfer tokens directly from victims who have granted approvals to the contract. This is possible because users approve the contract when interacting with liquidate and liquidateSmartCollateral. If the attacker transfers more loanToken than is needed for the liquidation, they profit from the surplus being returned to them. Note that the same issue is present in the existing flashLiquidate function.
Recommendations	Consider restricting these arbitrary calls, for example, by only allowing specific addresses to be targeted. It's likely not sufficient to simply block calls where the target equals the current loanToken, since an attacker could still call transferFrom on tokens unrelated to the current liquidation.
Comments / Resolution	Fixed by introducing a pairWhitelist, requiring that any address targeted in a swap must be explicitly whitelisted. Note that it is extremely important that this whitelist exclude any contracts that are themselves tokens, or any contracts in which the liquidator may hold value (e.g. balances, allowances, etc.). For example, whitelisting a UniswapV2 pair would be unsafe, as these contracts are LP tokens themselves, which could reintroduce the original issue if the LP token is approved to the liquidator by a victim. So, only general-purpose swap contracts, such as 1inch routers, should be whitelisted.

Issue_13	flashLiquidate and flashLiquidateSmartCollateral do not accrue interest
Severity	**Medium**
Description	The flashLiquidate and flashLiquidateSmartCollateral functions do not call moolah.accrueInterest before calculating the loanTokenAmountNeed for the liquidation. This differs from the liquidate and liquidateSmartCollateral functions. Since interest is not accrued, the loanTokenAmountNeed function can use out-of-date values relative to what Moolah will use in its calculations. In the worst case, this could lead to a revert if the calculated loanTokenAmount is too low.
Recommendations	Call accrueInterest at the start of the flashLiquidate and flashLiquidateSmartCollateral functions.
Comments / Resolution	Fixed as recommended.

Issue_14	flashLiquidateSmartCollateral does not return excess token0/token1
Severity	**Medium**
Description	At the end of flashLiquidateSmartCollateral, any excess loanToken is returned to msg.sender. There may also be leftover token0 or token1 from the LP redemption in onMoolahLiquidate, for instance if the token0Pair or token1Pair swaps do not consume the full amount of redeemed tokens. These tokens are not returned to the user and would remain behind in the contract. The user is expected to pass in swapToken0Data and swapToken1Data, which are swap calldatas and generally require specifying the exact amount of input tokens. However, the exact amount of input tokens are unknown to the caller at the time of the transaction creation, since it depends on the state of the LP and the returned values from the redeemLpCollateral function. Even minToken0Amt and minToken1Amt are not possible to predict perfectly, thus some slippage is expected. However if the user puts in minToken0Amt in the swap payload, they stand to lose any and all tokens above minToken0Amt since there are no refunds.
Recommendations	Consider adding logic to return leftover token0/token1 to the user.
Comments / Resolution	Fixed by adding functionality to return excess tokens.

Issue_15	Incorrect refund mechanism for loan token
Severity	Low
Description	The functions liquidate and liquidateSmartCollateral have a refund mechanism to transfer the remaining loan tokens to the caller after the liquidation. It works as follows: 1. Transfer loan tokens from caller to contract 2. Save balance (loanTokenBalanceBefore) 3. Do liquidation 4. Save balance again (collateralTokenBalanceAfter) 5. If the collateralTokenBalanceAfter > loanTokenBalanceBefore, then it transfers the difference to the caller. The issue with this mechanism is that it never works because loanTokenBalanceBefore should be saved before the actual transfer from the user. In the current implementation, the after balance will always be lower than the before balance because the liquidation always gets some loan tokens, but the leftovers will never actually be refunded.
Recommendations	It's recommended to move the variable loanTokenBalanceBefore before the actual transfer from the user. This way, the contract will actually refund the leftovers. On a side note, it seems like with the current implementation, there will never be any leftovers of loan tokens in the contract because the calculations at loanTokenAmountNeed are exactly the same as in Moolah. This causes Moolah to always transfer loanTokenAmount to execute the liquidation, so no loan tokens will be left at PublicLiquidator.
Comments / Resolution	Fixed as recommended.

Issue_16	Wrong OR operator on setMarketUserWhitelist.
Severity	**Low**
Description	When calling setMarketUserWhitelist, the function first checks either that the market is not already whitelisted on PublicLiquidator or that the liquidations are not permissionless on that Moolah market. *require(* *!marketWhitelist[id] ||* *!IMoolah(MOOLAH).isLiquidationWhitelist(id, address(0)),* *"market is already open for liquidate"* *);* However, if only one of the conditions is true, then there is no need to call setMarketUserWhitelist because users can already liquidate that market, either from PublicLiquidator or from Moolah itself.
Recommendations	It's recommended to update the check to ensure that both conditions must be false.
Comments / Resolution	Fixed as recommended.

Issue_17	liquidate does not work on smart collateral markets
Severity	**Informational**
Description	In theory, the liquidate function could be useful for a user wishing to liquidate a smart collateral position without redeeming the LP collateral tokens in the same step. This would be a useful feature considering the checkPriceDiff logic, which can lead to reverts during redemptions. However, it is not possible to use liquidate to liquidate smart collateral positions, since the LP collateral tokens are not transferable. Therefore, if a user wants to perform liquidations without redemptions happening in the same step, they would need to use a different contract or call liquidate on Moolah themselves.
Recommendations	Consider allowing liquidations without actual fund withdrawals or document this behavior.
Comments / Resolution	Fixed by adding a liquidateWithCollTransferOpt function, which works on both markets and can be done without redeeming the lp tokens. Note that this new feature of deferring the transfer of collateral tokens only supports smart collateral, all other liquidations must continue with the previous logic.

Issue_18	Liquidation functions do not allow wrapping native tokens
Severity	**Informational**
Description	The new liquidation functions allow the usage of native tokens, which can be part of the underlying stableswap pool. The issue is that the swap call simply sends these native tokens as msg.value to the pair/swap address. *(bool success,) = arb.token0Pair.call{value: _value}(arb.swapToken0Data);* The issue is that most swap protocols dont actually support native tokens, and only support wrapped native tokens (uni v3, pancakeswap CL pools etc). Thus users will be unable to interact with most liquidity pools in this manner.
Recommendations	Consider allowing wrapping the tokens before swapping.
Comments / Resolution	Acknowledged.

Issue_19	Possible reentrancy during liquidations
Severity	**Informational**
Description	During liquidations via liquidateSmartCollateral, the contract first pays out the tokens, which can be the native token, before doing the state change in postLiquidate. Thus the caller can get control of the flow before the state is updated completely. This violates the CEI pattern, allowing the caller to re-enter the contract in an incomplete state. However, there is no way to monetarily benefit from this.
Recommendations	Consider following the CEI pattern and doing the state change via postLiquidate before refunding the BNB tokens, or implementing nonReentrant modifiers in the whole contract.
Comments / Resolution	Fixed by adding reentrancy locks.

Issue_20	Add SafeTransferLib using directive for address type for better readability
Severity	**Informational**

Description	*using SafeTransferLib for address;*
	allows for better code readability wherever the library is used.
	For instance,
	SafeTransferLib.balanceOf(params.loanToken, address(this));
	becomes
	params.loanToken.balanceOf(address(this));

Recommendations	Add the using directive.
Comments / Resolution	Fixed as recommended.

Issue_21	Redundant payload in liquidateSmartCollateral
Severity	**Informational**
Description	The payload is decoded to the minimum amounts, but because swapSmartCollateral is disabled, these parameters are not used.
Recommendations	The payload can be commented out. *bytes memory /* payload */*
Comments / Resolution	Acknowledged.