OpenZeppelin | security

ListaDAO Moolah and StableSwap Audit

October 20, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-09-18 To 2025-10-03
Languages	Solidity

Total Issues	37 (19 resolved, 3 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	9 (4 resolved)
Low Severity Issues	8 (6 resolved, 1 partially resolved)
Notes & Additional Information	19 (8 resolved, 2 partially resolved)

Scope

OpenZeppelin audited the lista-dao/moolah repository at commit c2d4542.

In scope were the following files:

```
src
├── dex
│   ├── StableSwapFactory.sol
│   ├── StableSwapLP.sol
│   ├── StableSwapLPCollateral.sol
│   ├── StableSwapPool.sol
│   └── StableSwapPoolInfo.sol
├── moolah
│   └── Moolah.sol
└── provider
    └── SmartProvider.sol
```

System Overview

ListaDAO is a decentralized autonomous organization that develops and maintains various DeFi products on the (Binance Smart Chain) BSC network. The scope of this audit covers three core products developed by ListaDAO: the StableSwap automated market maker (AMM), the Moolah lending market, and the Smart Provider.

StableSwap is an AMM that implements a specialized curve for low-slippage trades between assets pegged 1:1, such as slisBNB/BNB or SlovBTC/BTCB. The implementation is a fork of PancakeSwap's StableSwap, which in turn is a Solidity port of Curve Finance's StableSwap Vyper contracts. Since StableSwap is designed for assets with stable prices, they mitigate impermanent loss and offer lower slippage compared to standard AMMs that use a constant product formula. To protect against extreme depegging events, ListaDAO has integrated an external price feed.

This mechanism, implemented in the `checkPriceDiff` function, compares the pool's internal price against an external price feed. If a user operation were to cause the price to deviate beyond the configured `price0DiffThreshold` or `price1DiffThreshold`, respectively, to the reference price feed, the transaction would revert. ListaDAO's implementation also diverges from PancakeSwap's by replacing its custom pausing and access-control mechanisms with OpenZeppelin's `Pausable` and `AccessControl` contracts.

The `Moolah` contract is a lending protocol forked from Morpho Blue's `Morpho` contract. It is designed as a primitive, monolithic lending protocol that features complete market isolation, where each market consists of a collateral asset and a borrowable asset. ListaDAO's implementation retains Morpho Blue's core logic but introduces several modifications, primarily by restricting its permissionless features. The key divergences are as follows:

- **Upgradability**: Unlike Morpho's immutable design, the `Moolah` contract is upgradable via the UUPS pattern.
- **Reentrancy Guard**: Market operations are protected against reentrancy attacks using OpenZeppelin's `ReentrancyGuard`.
- **Pausable Functionality**: The protocol can be paused, leveraging OpenZeppelin's `Pausable` contract.
- **Access Control**: Morpho's ownable feature has been replaced with a role-based access-control system using OpenZeppelin's `AccessControl`.

- **Restricted Market Creation**: Market creation can be restricted to an `OPERATOR` role, deviating from Morpho's permissionless model.
- **Per-Market Whitelists**: Optional per-market whitelists can restrict `supply`, `borrow`, and `supplyCollateral` functions to specific accounts.
- **Liquidator Whitelists**: An optional whitelist can be enabled for each market to restrict liquidations to authorized accounts.
- **Minimum Loan Value**: A `minLoanValue` state variable enforces a minimum value for deposits and borrows.
- **Oracle Semantics**: The `oracle` field in `MarketParams` struct now references a multi-asset oracle providing prices in USD, unlike Morpho's, which references a custom oracle that prices the collateral against the borrow asset. Consequently, price calculations are internalized in the `getPrice` function instead of being handled externally as in Morpho.
- **Providers**: A provider can be assigned to a market's loan asset, collateral asset, or both, granting it special permissions.
    - A **loan asset provider** has exclusive rights to borrow from the market on behalf of other accounts.
    - A **collateral asset provider** has exclusive rights to supply and withdraw collateral. It can initiate loans for other accounts, but can only withdraw collateral to its own address.
    - If a collateral provider is set, liquidations in that market trigger an external call to the provider's `liquidate` function, allowing for custom liquidation logic.
- **Default Market Fee**: The Moolah implementation includes a `DEFAULT_MARKET_FEE` of 5% for new markets.

The `SmartProvider` contract is a collateral-asset provider for Moolah. It allows users to atomically deposit liquidity into a StableSwap pool and use their LP position as collateral in a designated Moolah market. For this, the collateral asset is a wrapper token representing the StableSwap LP token on a 1:1 basis.

Security Model and Trust Assumptions

ListaDAO is assumed to be as secure as the reference implementations of PancakeSwap and Morpho for its StableSwap and Moolah implementations. All contracts of the system are upgradeable and contain a complex role-based access control configuration.

The following observations were made regarding the additional trust assumptions of the audited codebase:

- **Trusted Providers**: When a provider is assigned to a Moolah market for the loan or collateral asset, it bypasses the standard authorization checks for position ownership when borrowing the loan asset or withdrawing collateral. This allows the provider to borrow against or withdraw collateral from any user's position within that market. Since only the `MANAGER` role can assign providers, this role is trusted to only enable secure and vetted provider contracts.

Furthermore, if a collateral provider is set for a market, only that provider can supply or withdraw collateral. This could trap user funds if a provider is assigned to a market with existing user deposits. The ListaDAO team has acknowledged this and stated that for markets intended to be used with providers, the provider is set atomically at market creation to prevent this scenario. - **Oracle Precision**: Both `Moolah` and `StableSwapPool` contracts require oracles to provide prices with 8 decimal precision. In `StableSwapPool`, the `fetchOraclePrice` function hardcodes scaling factors based on this assumption. In `Moolah`, the `minLoanValue` parameter is denominated in USD and assumes that the underlying oracle prices also have 8 decimals. - **Atomic Proxy Initialization**: To prevent front-running attacks that could lead to unauthorized initialization of upgradable proxies, their deployment and initialization must be performed atomically within a single transaction. - **ERC-20 Token Compatibility**: The system is designed for standard-compliant ERC-20 tokens. It is assumed that any token integrated into the protocol has been carefully vetted for compatibility.

Privileged Roles

The OpenZeppelin team assumes that all privileged roles are trusted or controlled by the ListaDAO team and that the accounts used to authenticate in those roles are properly managed according to the different security and operational needs.

The following privileged roles are present across the audited contracts:

- `StableSwapFactory`:
  - `DEFAULT_ADMIN_ROLE`: Can manage roles, upgrade the contract, and call `addPairInfo` and `setImplsp`.
  - `DEPLOYER`: Can call `createSwapLP` and `createSwapPair`.
- `StableSwapLP`:
  - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.

    - `MINTER`: Can mint and burn LP tokens. This role is atomically granted to the corresponding StableSwap pool upon creation via `createSwapLP`.
- `StableSwapLPCollateral`:
    - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.
    - `MINTER`: Can mint and burn Collateral LP tokens. This role is atomically granted to the corresponding StableSwap pool upon creation via `createSwapLP`.
- `StableSwapPool`:
    - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.
    - `PAUSER`: Can pause contract functionality.
    - `MANAGER`: Can manage pool parameters, including fees, amplification parameter (`A`), and price difference thresholds. Can also withdraw admin fees and unpause the contract.
- `StableSwapPoolInfo`:
    - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.
- `Moolah`:
    - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.
    - `MANAGER`: Can manage protocol parameters, enable/disable Interest Rate Models (IRMs) and Loan-to-Value (LLTV) settings, set protocol fees and fee recipients, manage whitelists, configure providers, and unpause the contract.
    - `PAUSER`: Can pause contract functionality.
    - `OPERATOR`: Can create new markets if the role is populated.
    - `Provider`: A contract assigned to a market's loan or/and collateral asset, granting it special permissions. A loan asset provider has exclusive rights to borrow on behalf of users. A collateral asset provider has exclusive rights to manage collateral and can trigger custom liquidation logic.
    - `Whitelisted Accounts`: If enabled for a market, only these accounts can `supply`, `borrow`, and `supplyCollateral`.
    - `Whitelisted Liquidators`: If enabled for a market, only these accounts can perform liquidations.
- `SmartProvider`:
    - `DEFAULT_ADMIN_ROLE`: Can manage roles and upgrade the contract.
    - `Moolah`: The immutable `Moolah` contract address, which is the only address authorized to call the `liquidate` function.
    - `MANAGER`: Not used.

High Severity

H-01 Reentrancy in `supplyCollateral` Allows for Unbacked Collateral Minting

The `supplyCollateral` function of the `SmartProvider.sol` contract is vulnerable to a reentrancy attack. The function's logic calculates the amount of LP tokens to credit a user by measuring the contract's token balance before and after executing an external `safeTransferFrom` call. However, this function does not implement a reentrancy guard, making it susceptible to exploitation if a user interacts with it using a token that has built-in transfer hooks (e.g., ERC-777 tokens or custom ERC-20 implementations). As such, a malicious actor can leverage such a token to execute a recursive call back into the `supplyCollateral` function during the `safeTransferFrom` operation. This reentrant call can manipulate the state before the initial function call completes its logic, leading to an incorrect calculation of the collateral to be credited.

The direct impact of this vulnerability is that an attacker can artificially inflate their collateral balance within the Moolah protocol. By causing the balance calculation to be manipulated, an attacker can have the `SmartProvider` contract credit their position with more collateral tokens than are backed by the assets they actually deposited. This unbacked collateral can then be used to borrow legitimate assets from the lending market or be withdrawn by redeeming them for real assets from the underlying liquidity pool. In either scenario, the exploit leads to a direct and permanent loss of funds from the protocol.

It is recommended to protect all state-changing external functions in the `SmartProvider.sol` contract against reentrancy. This can be achieved by inheriting from OpenZeppelin's `ReentrancyGuardUpgradeable` contract and applying the `nonReentrant` modifier to the `supplyCollateral` function. To ensure comprehensive protection, this modifier should also be applied to all `withdrawCollateral` variants and any other public or external functions that are responsible for state modifications, thereby ensuring that their execution cannot be interrupted and manipulated by external calls.

Update: *Resolved in pull request #76 at commit 5b064bf.*

Medium Severity

M-01 `_isHealthyAfterLiquidate` Function Does Not Guarantee Position Health

The `Moolah` contract's `_liquidate` function is designed to support partial liquidations. To ensure protocol stability, a check is performed at the end of the transaction via the `_isHealthyAfterLiquidate` helper function. The explicit purpose and name of this function imply a guarantee: that any partial liquidation must be sufficient to restore the borrower's position to a healthy state, as defined by the market's loan-to-value ratio.

The implementation of the `_isHealthyAfterLiquidate` function contains a logical flaw that prevents it from fulfilling this purpose. The function checks if the borrower's remaining `borrowAssets` is greater than or equal to the `minLoan` value. If this condition is met, it immediately returns `true` without ever actually evaluating the position's collateral-to-debt ratio.

This flaw means the contract will fail to enforce that a position is solvent after a partial liquidation. While a competitive market of liquidators is economically incentivized to quickly liquidate any remaining unhealthy debt, this reliance on external actors is not guaranteed and may not hold under all market conditions. The primary issue is that the function's name and existence create a misleading on-chain promise of a safety guarantee that is not actually provided. This can mislead developers, auditors, and external integrators who rely on the contract's explicit logic, and it represents a significant deviation from the expected and stated behavior.

Consider refactoring the `_isHealthyAfterLiquidate` function to correctly reflect its purpose. The flawed shortcut checking against `minLoan` should be removed, and the function should be made to perform a full health check using the `_isHealthy` logic for all positions with remaining debt. This would ensure the function provides the on-chain guarantee that its name implies, removing ambiguity and the reliance on external economic assumptions for ensuring post-liquidation solvency.

Update: *Acknowledged, not resolved. The team stated:*

> *This upgrade will not include any modifications to the core contract that are unrelated to the smart collateral logic*

M-02 Fee Accrual to Zero Address Leads to Loss of Revenue

The Moolah protocol is designed to collect a portion of the interest paid by borrowers as a protocol fee. The `_accrueInterest` internal function calculates this revenue and allocates it to a designated `feeRecipient` address. This allocation occurs by minting new supply shares, representing a claim on the underlying assets, directly to the `feeRecipient`'s position within the market. This `feeRecipient` address is configurable and can be set by an account with the `MANAGER` role.

The `_accrueInterest` function does not validate the `feeRecipient` address before attempting to allocate the collected fees. If a valid recipient has not yet been configured, the `feeRecipient` address remains at its default value of `address(0)`. When interest is accrued under this condition, the function proceeds to mint the fee shares to the zero address.

Any supply shares minted to `address(0)` are irrecoverable, meaning all protocol revenue collected while the `feeRecipient` is unconfigured is permanently burned. Since interest accrual can be triggered by anyone through public functions like `supply` or `repay`, this creates a scenario where protocol revenue can be consistently lost until a manager sets a valid recipient address. This is especially critical in the period immediately following deployment before initial configuration is complete.

Consider adding a check to prevent fees from being allocated to the zero address. This can be accomplished either by requiring a non-zero `feeRecipient` to be provided during the contract's `initialize` function, or by modifying the `_accrueInterest` function to only mint fee shares if the `feeRecipient` is not `address(0)`.

Update: *Acknowledged, will resolve. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

M-03 Incorrect `DOMAIN_SEPARATOR` Construction Breaks EIP-712 Functionality

The `Moolah` contract utilizes the EIP-712 standard for secure, off-chain message signing, specifically for the `setAuthorizationWithSig` function. A critical component of this standard is the `DOMAIN_SEPARATOR`, a unique identifier that ensures a signature is valid only for a specific contract on a specific chain. In an upgradeable contract architecture, it is

essential that this separator is derived from the proxy's address, not the implementation contract's address, to prevent signature replay attacks across different instances.

The `DOMAIN_SEPARATOR` is declared as an `immutable` variable and is initialized within the `constructor` of the `Moolah` implementation contract. In a UUPS proxy pattern, the constructor of the implementation contract is executed only once at the time of its own deployment, not in the context of the proxy. Consequently, the value of `address(this)` used in the calculation is permanently set to the address of the implementation contract. Since `immutable` variables are written directly into the contract's bytecode, every proxy that points to this implementation will incorrectly use the same static domain separator based on the implementation's address.

This implementation error has two severe consequences. First, it renders the `setAuthorizationWithSig` feature entirely non-functional, as signatures correctly created off-chain using the proxy's address will always fail verification against the on-chain separator. Second, it introduces a cross-contract replay vulnerability: a signature generated for one proxy instance would be considered valid for any other proxy instance that uses the same implementation logic, allowing an attacker to replay an authorization grant across different deployments of the protocol.

Consider removing the `immutable` keyword from the `DOMAIN_SEPARATOR` state variable and moving its initialization from the `constructor` to the `initialize` function. By calculating the domain separator within the initializer, `address(this)` will correctly resolve to the proxy's address at runtime, ensuring that each proxy has a unique and correct separator. This will restore the intended functionality of EIP-712 features and mitigate the risk of cross-contract signature replay attacks.

Update: *Resolved in pull request #85.*

M-04 Duplicate Pair Creation Overwrites Existing Pool in Registry

The `StableSwapFactory` contract is designed to deploy and register unique `StableSwapPool` contracts for specific pairs of tokens. It maintains the mapping `stableSwapPairInfo`, which serves as a canonical on-chain registry for discovering the correct pool address for a given token pair. This registry is intended to enforce a one-to-one relationship between an ordered token pair and its corresponding swap contract.

The `createSwapPair` function does not validate whether a pool for the specified token pair already exists before proceeding with the creation and registration of a new one. An authorized

`DEPLOYER` can call this function multiple times for the same pair of tokens. Each subsequent call will deploy a new pool and LP token, and then unconditionally overwrite the existing entry in the `stableSwapPairInfo` mapping. This breaks the one-to-one invariant of the registry, leading to several negative consequences:

- The original pool becomes undiscoverable through the factory's `getPairInfo` function, potentially stranding liquidity.
- The `pairLength` counter becomes an inaccurate representation of unique pairs.
- Duplicate `NewStableSwapPair` events can mislead off-chain indexers and user interfaces.

Consider adding a check at the beginning of the `createSwapPair` function to ensure that a pool for the given token pair has not already been created. The function should read from the `stableSwapPairInfo` mapping and revert if an entry for the token pair already exists. This will enforce the intended one-to-one relationship between a token pair and its pool, preventing state inconsistencies and ensuring the integrity of the on-chain registry.

Update: *Resolved in pull request #84 at commit fcd2909 and pull request #76 at commit b314a87.*

M-05 Improper Role Initialization Grants Admin Privileges

The `StableSwapFactory` contract defines a `DEPLOYER` role intended for addresses that are authorized to create new swap pairs. This role is meant to be distinct from the `DEFAULT_ADMIN_ROLE`, which holds higher-level administrative privileges, such as the ability to set the implementation addresses for new contracts. This separation is designed to follow the principle of least privilege, ensuring that pair creators do not have unnecessary administrative power.

The `DEPLOYER` role identifier is declared as a public state variable instead of a `constant`. In an upgradeable proxy architecture, state variables initialized at declaration are set in the implementation contract's constructor but remain at their default value (zero) in the proxy's storage. Within OpenZeppelin's `AccessControl` framework, the `bytes32(0)` value is the identifier for the `DEFAULT_ADMIN_ROLE` role. Consequently, when the `initialize` function calls grants the `DEPLOYER` role, it is effectively granting the `DEFAULT_ADMIN_ROLE` to each deployer address. This results in a privilege escalation, where accounts intended to have limited creation permissions are instead given full administrative control over the factory.

Consider declaring the `DEPLOYER` role identifier as a `constant` instead of a state variable. By changing the declaration to a constant, its value will be compiled directly into the contract's bytecode and will be correctly referenced in the proxy's context. This will ensure that the `_grantRole` function assigns the intended, limited `DEPLOYER` role and prevents the unintended privilege escalation to `DEFAULT_ADMIN_ROLE`.

Update: *Resolved in pull request #76 at commit d7560fe.*

M-06 Admin Role on LP Token Can Be Permanently Lost During Pair Creation

The `createSwapPair` function of the `StableSwapFactory` contract contains a flaw in its administrative-role-transfer logic for newly created LP tokens. The function correctly initializes the LP token with the factory contract as its sole administrator, but then proceeds to grant the admin role to a user-specified address and unconditionally revokes its own admin role. If a user specifies the factory contract itself as the new administrator, the function will grant the role to itself (a redundant action) and then proceed to revoke its own, and only, admin role, leaving the LP token without any administrator.

This could lead to a permanent and irreversible loss of administrative control over a newly created LP token contract. If the admin role is burned in the manner described, all functionality requiring this role, including the critical ability to upgrade the contract's implementation via its proxy, will be lost forever. This represents a significant operational risk, as it allows for a simple misconfiguration to render a core component of the DEX pair permanently unmanageable and non-upgradeable.

Consider adding a preventative validation check within the `createSwapPair` function. This check should ensure that the address provided as the new administrator for the LP token is not the same as the `StableSwapFactory` contract's own address. The transaction should be reverted if these addresses match, thereby preventing any scenario where the administrative role could be accidentally destroyed during the creation process.

Update: *Resolved in pull request #84 at commit c53391f.*

M-07 Liquidation Can Create Dust Positions Below Minimum Loan Size

The `Moolah` contract establishes a minimum loan size to prevent the creation of economically insignificant dust positions. This minimum is enforced within functions such as `borrow` and `repay` via the `_checkBorrowAssets` function, which ensures that a user's remaining debt is either zero or greater than or equal to the configured `minLoan` value. The liquidation mechanism, however, contains separate logic within the `_isHealthyAfterLiquidate` function to validate the state of a position after a partial liquidation has occurred.

The `_isHealthyAfterLiquidate` function, which is called at the end of a liquidation, does not uphold this minimum loan invariant. If a partial liquidation reduces a borrower's debt to an amount greater than zero but less than the `minLoan` value, the function does not revert. Instead, it proceeds to a standard health factor check. If the position is deemed healthy at this new, smaller debt level, the liquidation transaction is allowed to succeed. This creates an inconsistency where the liquidation process can result in a dust position, a state which other core functions of the protocol are designed to prevent.

Consider modifying the `_isHealthyAfterLiquidate` function to enforce the same minimum loan size invariant that is applied elsewhere in the protocol. The function should ensure that after a partial liquidation, the borrower's remaining `borrowAssets` is either zero or greater than or equal to the value returned by `minLoan`. This would align the behavior of the liquidation mechanism with other core functions and prevent positions from entering an inconsistent dust state.

Update: *Acknowledged, not resolved. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

M-08 Minimum Supply Invariant Can Be Bypassed via Withdrawal

The `Moolah` contract establishes a minimum loan size to prevent the creation of economically insignificant dust positions, which can bloat state and be inefficient to manage. This invariant is enforced for borrowers in both the `borrow` and `repay` functions via the `_checkBorrowAssets` function. A similar check, `_checkSupplyAssets`, is used in the `supply` function to ensure that new or augmented supply positions also adhere to this minimum threshold.

While the `minLoan` invariant is enforced when users supply assets, the corresponding check is absent from the `withdraw` function. The `withdraw` function allows a user to remove any portion of their supplied assets without verifying that their remaining supply balance, if not zero, is still greater than or equal to the `minLoan` value. This creates an asymmetry in the protocol's logic and allows the dust-prevention rule to be bypassed. A user can supply an amount greater than the minimum, and then immediately call `withdraw` to remove most of it, leaving a dust position that violates the protocol's intended invariant.

Consider enforcing the minimum supply invariant within the `withdraw` function to mirror the logic used for repayments. A call to `_checkSupplyAssets` should be added before the function concludes to ensure that a user's remaining supply position is either fully withdrawn or remains above the `minLoan` threshold. This will prevent the creation of dust supply positions and ensure consistent application of the protocol's invariants across all core functions.

Update: *Acknowledged, not resolved. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

M-09 Bad-Debt Socialization Can Be Maliciously Skipped

In the `Moolah` contract, the `_liquidate` function is meant to allow liquidators to repay borrowers' loans at a discount, given by a liquidation incentive factor, deleting their unhealthy positions and in exchange receiving their collateral.

Liquidators can choose the exact amount of collateral they want to seize, which translates to a specific amount of debt to repay, or the exact amount of debt shares to repay, which also translates to a certain amount of collateral to seize. It is up to them to decide whether to perform a partial or a full liquidation by customizing the `seizedAssets` or the `repaidShares` parameters.

A liquidatable position may sometimes generate bad debt, which is debt that will never be repaid by anyone. When this happens, the loss is socialized among all suppliers equally.

Consider a scenario where there exists a market with the following configuration:

- WETH is the collateral token of choice.
- USDC is the borrowable token.
- LLTV of 80%

- Incentive factor of 6%.
- Alice supplies 5,000 USDC, and she will be the victim.
- Bob supplies 5,000 USDC, and he will be the attacker.
- Charlie deposits 4 ETH as collateral and borrows the maximum amount available of 6400 USDC.
- ETH price is $2,000.

If the price of ETH suddenly drops to 1,500 USDC, Charlie's position becomes unhealthy, leaving bad debt behind even if fully liquidated. A liquidator will notice this unhealthy position and go ahead and liquidate it by seizing the full 4 ETH collateral from Charlie and repaying 5240 USDC worth of debt, calculated in `repaidAssets`. The remaining 760 USDC is socialized evenly among existing suppliers.

Consider an alternative scenario where Bob, one of the existing suppliers, realizes that there is going to be some bad debt generated by this position and wants to escape its socialization. He decides to go ahead and liquidate Charlie's position himself, by seizing exactly the full position collateral minus 1 wei. When doing so, the final collateral on Charlie's position will be exactly 1 wei, bypassing the condition looking for bad debt to socialize.

At this point, Bob decides to use the liquidation callback to perform the following actions:

- Withdraw his supplied amount. There is enough liquidity to do so and no bad debt has been socialized yet.
- Call liquidate on Charlie's position again, using 1 wei as the value for `seizedAssets`. This time, the liquidation fully removes Charlie's collateral and, thus, the entire bad debt is socialized among all suppliers. However, Bob is not a supplier anymore, so the entire bad debt is allocated to Alice.
- Now that the bad-debt socialization has been successfully applied, Bob decides to resupply the same amount of USDC he had initially, leaving his position unaltered by the bad-debt socialization. Alice has suffered the entirety of the loss, so Charlie has effectively managed to steal USDC from Alice by redirecting the loss towards her.

This attack was made possible because the current implementation only socializes debt when a position is fully liquidated, but not on partial liquidations.

Consider revisiting the design of the liquidation mechanism and deciding whether both liquidations should be allowed. If partial liquidations are allowed, consider the possibility of partial bad-debt socialization on every liquidation, provided that the position is so unhealthy that it will generate bad debt in any case at current market prices.

Update: *Acknowledged, not resolved. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

Low Severity

L-01 Incomplete Access Control on `StableSwapLPCollateral` Transfers

The `StableSwapLPCollateral` contract represents a user's collateral position within the Moolah lending protocol. It is an ERC-20 token whose transfers are intended to be restricted. To enforce this, the contract defines an `onlyMoolah` modifier, which is designed to ensure that only the main `Moolah` contract can initiate transfers of the token, for example during liquidations or collateral withdrawals.

The `onlyMoolah` modifier is applied only to the `transfer` function, while the standard `transferFrom` function inherited from `ERC20Upgradeable` remains unrestricted. While it is understood that under normal operating conditions this token is only intended to be held by the `Moolah` contract (i.e., being atomically minted by the `SmartProvider` and supplied as collateral), this assumption is not enforced on-chain. Should a user ever come into possession of these tokens, they could bypass the intended transfer restriction by calling `approve` and having a third party use the unrestricted `transferFrom` function to move the tokens.

Consider either removing the `onlyMoolah` modifier if the restriction is deemed unnecessary, or applying the control consistently to all transfer mechanisms. To enforce the restriction properly, the internal `_transfer` function from the underlying `ERC20` contract should be overridden with the desired access-control logic. Since both the public `transfer` and `transferFrom` functions rely on `_transfer` internally, this change would ensure that access control is correctly and comprehensively applied to all token movements.

Update: *Resolved in pull request #86.*

L-02 Missing Token Sorting in `addPairInfo` Leads to Inconsistent State

The `StableSwapFactory` contract serves as a registry for swap pools, using the mapping `stableSwapPairInfo` to link a pair of token addresses to a specific pool contract. To

ensure a canonical key for each pair regardless of the order in which token addresses are provided, the factory uses a `sortTokens` function. Core functions like `createSwapPair` and the public getter `getPairInfo` consistently use this sorting logic to enforce a predictable storage layout where `token0` is always less than `token1`.

The `addPairInfo` function, which allows an admin to manually register an existing pool, does not adhere to this sorting convention. It reads the token addresses directly from the provided swap contract and uses them as keys for the `stableSwapPairInfo` mapping without first passing them through the `sortTokens` function. If the swap contract happens to store its token addresses in a non-canonical order (i.e., it was created by a different factory), the pair information will be written to an incorrect location in the mapping. This will cause subsequent lookups for that pair via `getPairInfo` to fail, as `getPairInfo` will search for the sorted key. This inconsistency can lead to a fragmented and unreliable registry, and could allow for duplicate pairs to be registered.

Consider calling the `sortTokens` function within `addPairInfo` before the token addresses are used to update the `stableSwapPairInfo` mapping. This will ensure that manually registered pairs adhere to the same canonical ordering as that of the pairs created via the `createSwapPair` function, guaranteeing the consistency and reliability of the factory's on-chain registry.

Update: *Resolved in pull request #84 at commit 9dde20d.*

L-03 `get_virtual_price` Function Reverts When LP Token Supply Is Zero

The `StableSwapPool` contract includes a `get_virtual_price` `view` function that is intended to provide a key metric for the pool's performance. This function calculates the value of a single LP token in terms of the underlying assets, scaled to a common precision. This value is often used by off-chain analytics, user interfaces, and other on-chain protocols to track the growth of the LP token value over time.

The `get_virtual_price` function calculates its value by dividing the pool's invariant by the `totalSupply` of the LP token. The function does not handle the edge case where the `totalSupply` is zero, a state that occurs when the pool is newly deployed and has no liquidity. Consequently, any call to `get_virtual_price` while the pool is empty will result in a division-by-zero error, causing the call to revert. While the `Moolah` and `SmartProvider` contracts do not appear to use this function, it will fail for any client that does.

Consider adding a check to the `get_virtual_price` function to handle the case where the LP token supply is zero.

Update: *Resolved in pull request #76 at commit cec91b7.*

L-04 Division by Zero in `view` Function Leads to Denial of Service

The `StableSwapPoolInfo` contract provides `view` functions to calculate the expected outcomes of interacting with a `StableSwapPool`. One such function, `calc_coins_amount`, is used to determine the amount of underlying tokens corresponding to a given amount of LP tokens. The `SmartProvider` contract relies on this function for its own `peek` oracle functionality, which is in turn used by the main `Moolah` contract to price the LP token collateral and assess the health of user positions.

The `calc_coins_amount` function calculates the amount of underlying tokens by dividing by the LP token's `totalSupply`. However, the function does not handle the edge case where the `totalSupply` is zero, which occurs when a `StableSwapPool` is newly deployed and has not yet received liquidity. In this state, any call to `calc_coins_amount` will attempt a division by zero, causing the call to revert.

Consider adding a check at the beginning of the `calc_coins_amount` function in `StableSwapPoolInfo`. If the `totalSupply` of the LP token is zero, the function should gracefully return a zero-filled array instead of proceeding with the division. This will prevent the function from reverting on an empty pool and ensure that dependent oracle functions, such as `SmartProvider.peek`, can return a sensible default value, thereby preventing the denial of service on the core `Moolah` contract.

Update: *Resolved in pull request #76 at commit 3aea17e.*

L-05 `setImpls` Function Unnecessarily Couples Implementation Updates

The `StableSwapFactory` contract is responsible for deploying new swap and LP token contracts based on the master implementation addresses, `lpImpl` and `swapImpl`. The `setImpls` function provides an administrative interface for an address with the `DEFAULT_ADMIN_ROLE` to update these two implementation addresses, allowing the factory to deploy new pairs using updated contract logic in the future.

The `setImpls` function requires both the `_newLpImpl` and `_newSwapImpl` parameters to be non-zero within the same transaction, due to a combined `require` statement. This design improperly couples the update of two independent implementation addresses. It prevents an administrator from updating only one of the two addresses in a single call, forcing them to re-supply the current, unchanged address for the other parameter. This inflexibility makes the function brittle and can complicate administrative procedures, especially in potential recovery scenarios.

Consider decoupling the update logic for the two implementation addresses. This could be achieved by replacing the single `setImpls` function with two separate administrative functions, such as `setLpImpl` and `setSwapImpl`, each responsible for updating only one address. This approach would provide greater flexibility, simplify administrative workflows, and adhere more closely to the single-responsibility principle for function design.

Update: *Resolved in pull request #84 at commit 8e167e2.*

L-06 Missing Zero-Address Checks

When operations with address parameters are performed, it is crucial to ensure the address is not set to zero. Setting an address to zero is problematic because it has special burn/renounce semantics. This action should be handled by a separate function to prevent accidental loss of access during value or ownership transfers.

Throughout the codebase, multiple instances of missing zero-address checks were identified:

- In the `addLiquidationWhitelist` function, no zero-address check is performed on the `account` parameter to ensure the liquidation whitelist is working as intended.
- In the `addWhitelist` function, no zero-address check is performed on the `account` parameter to ensure the whitelist is working as intended.
- in the `enableIrm` function, no zero-address check is performed on the `irm` parameter to ensure the interest rate model is working as intended.
- No zero-address check is performed upon granting admin rights to the `admin` address in the `initialize` function of the `StableSwapPoolInfo` contract.
- No zero-address check is performed upon granting admin rights to the `_admin` address to the newly created `StableSwapPool` in the `createSwapPair` function of the `StableSwapFactory` contract.
- No zero-address check is performed on the `_tokenA` and `_tokenB` parameters of the `createSwapLP` function of the `StableSwapFactory` contract.

Consider performing a zero-address check before assigning a state variable.

Update: *Partially Resolved in pull request #76 at commit 694a548. The team stated:*

> *Add zero address check in `StableSwapPoolInfo.initialize`*

L-07 Inconsistent Use of `_msgSender()` and `msg.sender`

In Solidity, `msg.sender` refers to the immediate caller of a function. To support meta-transaction systems which allow users to have transactions relayed, contracts can use the `_msgSender()` helper function. When properly configured, `_msgSender()` returns the address of the original user who signed the message, while `msg.sender` would be the address of the relaying contract. Consistent use of `_msgSender()` throughout a protocol is essential for meta-transactions to function correctly across complex, multi-contract interactions.

The codebase does not apply a consistent strategy for retrieving the sender's address. The `StableSwapLPCollateral` contract uses `_msgSender()` within its overridden `transfer` function, indicating a potential intent to support meta-transactions. However, nearly all other contracts in the system, including the core `Moolah`, `StableSwapFactory`, and `StableSwapPool` contracts, exclusively use the global `msg.sender` variable for all authorization checks and business logic. This inconsistency means that if meta-transactions are an intended feature, they will fail when a relayed call reaches a contract that relies on `msg.sender`, as it will incorrectly identify the relayer as the user. If meta-transactions are not intended, the use of `_msgSender()` is unnecessary and may cause confusion.

Consider establishing a consistent, protocol-wide strategy for handling the transaction sender. If meta-transactions are a desired feature, all relevant contracts should be updated to inherit from a context-aware base contract and use `_msgSender()` in place of `msg.sender` for all user-facing authorization and logic. If meta-transactions are not intended to be supported, the use of `_msgSender()` should be removed and replaced with `msg.sender` for consistency and clarity.

Update: *Acknowledged, not resolved. The team stated:*

> *We'd like to use the same code base as with `ERC20Upgradeable.sol`*

L-08 Redundant Slippage Check Increases Gas Costs

The `SmartProvider` contract acts as a wrapper for the `StableSwapPool` DEX, enabling users to deposit and withdraw liquidity. To protect users from slippage, functions such as `supplyCollateral` and the various withdrawal methods implement checks to ensure that the final transaction outcome meets user-specified limits. These checks are critical for mitigating negative price movements and potential front-running attacks during the execution of a transaction.

Throughout the `SmartProvider` contract, a three-part validation strategy is used. First, a pre-flight check simulates the transaction using the `StableSwapPoolInfo` helper contract to fail early if the request is already unachievable, saving user gas and improving the user experience. Second, the primary on-chain check is performed by passing the user's slippage parameters directly to the `StableSwapPool` contract, which provides the core security against slippage during execution.

Third, a post-flight check is conducted by the `SmartProvider` itself, which measures its token balances before and after the DEX interaction to verify the outcome. While the pre-flight and on-chain checks serve distinct and valuable purposes for UX and security, respectively, the final post-flight check is redundant, as it re-validates an outcome that should have already been guaranteed by the `StableSwapPool`'s own internal validation. The current multi-layered approach provides robust validation but incurs additional gas costs due to the post-flight check. The pre-flight check is a valuable optimization and should be retained. The on-chain check within the `StableSwapPool` is the essential security guarantee.

Therefore, to optimize gas consumption while maintaining security, consider removing the final, post-flight balance verification from functions like `supplyCollateral`, `_redeemLp`, and the other withdrawal methods. This would reduce the gas cost of every interaction by eliminating extra `balanceOf` calls and `require` statements, placing trust in the `StableSwapPool`'s on-chain validation as the single source of truth for slippage protection.

Update: *Resolved in pull request #83.*

Notes & Additional Information

N-01 Redundant `contains` Check Before Modifying `EnumerableSet`

The `Moolah` contract uses OpenZeppelin's `EnumerableSet` library to manage access-control lists, such as `liquidationWhitelist` and the general `whiteList`. Functions like `addLiquidationWhitelist` and `removeLiquidationWhitelist` are used by administrators to modify the membership of these sets, ensuring that only authorized accounts can perform certain actions.

The functions for adding to and removing elements from these sets first perform an explicit `contains` check within a `require` statement, and only then call the corresponding `add` or `remove` function. This pattern is redundant because the `add` and `remove` functions from OpenZeppelin's `EnumerableSet` library already perform this internal existence check and return a boolean value indicating whether the state had been changed. This results in two storage reads for the same check where only one is necessary, leading to unnecessary gas consumption for each administrative action.

Consider refactoring the whitelist management functions to remove the explicit `contains` call and instead using the boolean return value of the `add` and `remove` functions directly in the `require` statement.

Update: *Acknowledged, will resolve. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

N-02 Redundant Zero-Address Check in `addProvider`

In the `Moolah` contract, the `addProvider` function contains a check to prevent the zero address from being added. However, this check is redundant and will never be triggered. The first line performs an external call to the `provider` address. If the `provider` address were `address(0)`, this external call would fail and cause the transaction to revert before the

explicit zero-address check is ever reached. Therefore, the check provides no additional security and constitutes dead code.

Consider removing the redundant check from the `addProvider` function. Removing this line of code will save a small amount of gas on each successful call to `addProvider` and will eliminate the dead code from the contract.

Update: *Acknowledged, will resolve. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

N-03 Redundant Market ID Calculation in Helper Functions

The `_checkSupplyAssets`, `_checkBorrowAssets`, and `_isHealthyAfterLiquidate` helper functions are passed the full `marketParams` struct as an argument and proceed to recalculate the market `Id` internally. This calculation is redundant because the parent functions `supply`, `borrow`, and `repay` have already computed the exact same `Id` earlier in the same transaction. Performing this hash operation multiple times within a single call stack results in unnecessary gas consumption.

Consider refactoring the internal helper functions to accept the pre-calculated market `Id` as a direct argument instead of, or in addition to, the `marketParams` struct. The parent functions can then pass the `Id` they have already computed, eliminating the redundant hash calculation in the helpers to save gas costs.

Update: *Acknowledged, will resolve. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

N-04 Duplicate Storage Read in `_liquidate`

In the `_liquidate` function of the `Moolah` contract, the `providers[id][marketParams.collateralToken]` storage slot is read twice in close succession. It is first accessed within a `require` statement and then immediately accessed again to be assigned to a local variable. As each storage read incurs a gas cost, this pattern of reading the same value twice is inefficient.

Consider optimizing gas usage by caching the provider address into a local variable once at the beginning of the logic block.

Update: *Acknowledged, will resolve. The team stated:*

> *Will revise in a different issue raised by other audit report*

N-05 Unused `MANAGER` Role

The `SmartProvider` contract defines and grants a `MANAGER` role during its `initialize` function as part of its access-control scheme. However, this `MANAGER` role is never actually used to protect any function or perform any administrative action. The presence of this unused role and its associated setup logic adds unnecessary complexity, increases deployment gas costs, and can be misleading as it implies a set of permissions that do not exist.

Consider removing the `MANAGER` role and all associated logic from the `SmartProvider` contract.

Update: *Resolved in pull request #76 at commit b29b4a2.*

N-06 Inconsistent Syntax for Error Messages

The codebase does not follow a single, consistent pattern for handling error messages. In several places, such as the `SmartProvider` contract, there is a mix of two styles: some `require` statements use custom error constants imported from a library, while others use hardcoded string literals.

Consider adopting a single, consistent strategy for error messaging throughout the entire codebase. Using a centralized error library, as is already done in parts of the project, is a common best practice. Refactoring all `require` statements to use constants from the `ErrorsLib` library instead of string literals would lead to a more maintainable, readable, and gas-efficient codebase.

Update: *Acknowledged, will resolve.*

N-07 Typographical Errors

Throughout the codebase, two instances of typographical errors were identified:

- 'conracts' instead of 'contracts' in the Docstring of the `createSwapLP` function
- 'conracts' instead of 'contracts' in the Docstring of the `createSwapPair` function

Consider fixing any instances of typographical errors to improve the readability of the codebase.

Update: *Resolved in pull request #84 at commit b6102a8 and pull request #76 at commit 6fe37e3.*

N-08 Unnecessary Allowance Granted for Role-Based `burnFrom`

In the `SmartProvider` contract, the `withdrawCollateralImbalance` and `withdrawCollateralOneCoin` functions grant an ERC-20 allowance to the `StableSwapPool` before calling the respective function to remove liquidity. This is done in anticipation of the `StableSwapPool` contract needing to spend the `SmartProvider` contract's LP tokens. However, the `burnFrom` function of the `StableSwapLP` token contract is a custom, access-controlled function restricted by an `onlyMinter` modifier. As the `StableSwapPool` contract is designated as the sole `minter` during its creation, it is authorized to call `burnFrom` directly without requiring any ERC-20 allowance. The allowance granted by `SmartProvider` is therefore never checked, making the `safeIncreaseAllowance` calls unnecessary.

Consider removing the `safeIncreaseAllowance` calls from the `withdrawCollateralImbalance` and `withdrawCollateralOneCoin` functions of the `SmartProvider` contract. Since the `StableSwapPool`'s authority to burn LP tokens is managed via a dedicated `minter` role instead of the standard ERC-20 allowance mechanism, these calls serve no purpose and only add to the gas cost of withdrawal transactions.

Update: *Resolved in pull request #76 at commit 264ce5d.*

N-09 Use of `abi.encodeWithSignature` Is Not Type-Safe

The `StableSwapFactory` contract uses `abi.encodeWithSignature` to construct this initialization calldata in both the `createSwapLP` and `_createSwapPair` functions. This function takes a manually-provided function signature as a string and does not perform compile-time checks to ensure that the supplied arguments match the types in the signature string. This approach is brittle and error-prone. If the signature of an `initialize` function in an implementation contract is ever changed, the corresponding string in the factory must also be updated manually. A failure to do so would not be caught by the compiler and would result in the factory deploying proxies with incorrect calldata, causing all new deployments to fail.

Consider replacing the use of `abi.encodeWithSignature` with the type-safe `abi.encodeCall`. This function takes a function pointer and its arguments, and the Solidity compiler will verify at compile time that the arguments match the function's signature. Migrating to `abi.encodeCall` would make the factory contract more robust against future changes and prevent a class of potential deployment failures by enforcing type safety on the initialization calldata.

Update: *Acknowledged, will resolve.*

N-10 Unaddressed TODO Comments

During development, having well described TODO comments will make the process of tracking and resolving them easier. However, these comments might age and important information for the security of the system might be forgotten by the time it is released to production.

In the `StableSwapFactory` contract, a TODO comment was identified.

Consider removing all instances of TODO comments and instead tracking them in the issues backlog. Alternatively, consider linking each inline TODO to a corresponding backlog issue.

Update: *Resolved in pull request #84 at commit 4f2c064.*

N-11 `createSwapLP` Function Has Unnecessarily Public Visibility

The `createSwapLP` function is declared with `public` visibility, allowing it to be called directly by any account with the `DEPLOYER` role. However, the function is only intended to be

used as an internal helper for `createSwapPair`. If called directly, it deploys an LP token contract but leaves its `admin` and `minter` roles assigned to the factory contract itself. The factory has no functions to exercise these administrative or minting capabilities, and the subsequent setup steps within `createSwapPair` are skipped. This results in the creation of an incompletely configured and effectively non-functional LP token.

Consider changing the visibility of the `createSwapLP` function from `public` to `internal`. Since the function's logic is tightly coupled with the setup process in `createSwapPair` and it does not produce a fully functional contract when called in isolation, restricting its visibility will prevent it from being used as an incorrect entry point. This change would enforce the intended creation process and reduce the risk of deploying misconfigured contracts.

Update: *Resolved in pull request #76 at commit 7b2ce25 and pull request #76 at commit 6fe37e3.*

N-12 Redundant `sortTokens` Call in `_createSwapPair`

The `StableSwapFactory` contract uses a `sortTokens` function to ensure that token pairs are handled in a consistent, ordered manner, regardless of the order in which they are supplied as arguments. The public `createSwapPair` function correctly uses this sorting logic before passing the tokens to the internal `_createSwapPair` function, which is responsible for deploying the swap contract proxy.

The internal `_createSwapPair` function also uses `sortTokens` on the token addresses it receives. Since this function is only ever called by the `createSwapPair` function, which has already sorted the tokens, this second call to `sortTokens` is redundant. The token addresses are guaranteed to be in the correct order before `_createSwapPair` is invoked, so the second sorting operation provides no additional benefit and consumes a small, unnecessary amount of gas in every pair-creation transaction.

Consider removing the redundant `sortTokens` call from the `_createSwapPair` function. This is because the function's only caller, `createSwapPair`, already guarantees that the token addresses are sorted.

Update: *Resolved in pull request #76 at commit 0a99ca3.*

N-13 Incomplete Docstrings

Throughout the codebase, multiple instances of incomplete docstrings were identified:

- In the `createSwapLP` function of `StableSwapFactory.sol`, not all return values are documented.
- In the `createSwapPair` function of `StableSwapFactory.sol`, the `_name`, `_symbol`, `_admin`, `_manager`, `_pauser` and `_oracle` parameters are not documented. In addition, not all return values are documented.
- In the `transfer` function of `StableSwapLPCollateral.sol`, the `to` and `value` parameters are not documented. In addition, not all return values are documented.
- In the `initialize` function of `StableSwapPool.sol`, the `_oracle` parameter is not documented.
- In the `exchange` function of `StableSwapPool.sol`, the `i`, `j`, `dx`, and `min_dy` parameters are not documented.
- In the `get_coins_amount_of` function of `StableSwapPoolInfo.sol`, not all return values are documented.
- In the `initialize` function of `Moolah.sol`, the `_minLoanValue` parameter is not documented.
- In the `isHealthy` function of `Moolah.sol`, the `marketParams`, `id` and `borrower` parameters are not documented. In addition, not all return values are documented.
- In the `addProvider` function of `Moolah.sol`, the `id` and `account` parameters are not documented.
- In the `removeProvider` function of `Moolah.sol`, the `id` and `provider` parameters are not documented.

Consider thoroughly documenting all functions/events (and their parameters or return values) that are part of a contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Partially Resolved in pull request #76 at commit 6c79cc4.*

N-14 Over-Privileged `DEFAULT_ADMIN_ROLE` Violates Principle of Least Privilege

The audited contracts employ OpenZeppelin's `AccessControl` pattern for managing permissions, which allows for the creation of distinct roles for different administrative functions. Security best practices dictate that the `DEFAULT_ADMIN_ROLE` should be used exclusively as

a 'role-manager-role' for granting and revoking other, more specific operational roles. Critical permissions, such as upgrading contracts, pausing functionality, or changing key parameters, should be assigned to separate, dedicated roles to ensure a proper separation of duties.

Throughout the codebase, this principle is not consistently followed. The `DEFAULT_ADMIN_ROLE` is frequently granted powerful operational permissions beyond role management, effectively making it a superuser. For instance, in the `Moolah` contract and other upgradeable contracts, this role has the authority to authorize implementation upgrades. In the `StableSwapFactory`, it is used to manage pair information, and in the LP token contracts, it may be used to control minting rights. This overloading of a single role with multiple, highly sensitive responsibilities creates a centralized point of failure.

Consider conducting a system-wide review of the access control implementation to enforce a strict separation of duties. A dedicated `UPGRADER_ROLE` should be created for all upgradeable contracts, and other critical operational permissions should also be moved to distinct, specific roles. The `DEFAULT_ADMIN_ROLE` should then be restricted solely to the administration of these other roles. This refactoring would align the project with security best practices and substantially reduce the attack surface by ensuring that no single role holds a dangerous combination of administrative and operational powers.

Update: *Acknowledged, will resolve. The team stated:*

> *We will transfer DEFAULT_ADMIN_ROLE to a TimeLock contract.*

N-15 Use `calldata` Instead of `memory`

When dealing with the parameters of `external` functions, it is more gas-efficient to read their arguments directly from `calldata` instead of storing them to `memory`. `calldata` is a read-only region of memory that contains the arguments of incoming `external` function calls. This makes using `calldata` as the data location for such parameters cheaper and more efficient compared to `memory`. Thus, using `calldata` in such situations will generally save gas and improve the performance of a smart contract.

Throughout the codebase, multiple instances where function parameters should use `calldata` instead of `memory` were identified:

- In `StableSwapFactory.sol`, the `_name` parameter
- In `StableSwapFactory.sol`, the `_symbol` parameter
- In `StableSwapLP.sol`, the `name` parameter
- In `StableSwapLP.sol`, the `symbol` parameter

- In `StableSwapLPCollateral.sol`, the `_name` parameter
- In `StableSwapLPCollateral.sol`, the `_symbol` parameter
- In `StableSwapPool.sol`, the `amounts` parameter
- In `StableSwapPool.sol`, the `amounts` parameter
- In `StableSwapPool.sol`, the `min_amounts` parameter
- In `StableSwapPool.sol`, the `amounts` parameter
- In `StableSwapPoolInfo.sol`, the `amounts` parameter
- In `StableSwapPoolInfo.sol`, the `amounts` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `ids` parameter
- In `Moolah.sol`, the `accounts` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `payload` parameter
- In `Moolah.sol`, the `authorization` parameter
- In `Moolah.sol`, the `marketParams` parameter
- In `Moolah.sol`, the `marketParams` parameter

Consider using `calldata` as the data location for the parameters of `external` functions to optimize gas usage.

Update: *Partially Resolved in pull request #76 at commit b6e5b6e. The team stated:*

> *Revised as per suggested for all contracts except `Moolah`.*

N-16 Missing Security Contact

Providing a specific security contact (such as an email address or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice is quite beneficial as it permits the code owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition,

if the contract incorporates third-party libraries and a bug surfaces in those, it becomes easier for their maintainers to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, multiple instances of contracts not having a security contact were identified:

- The `StableSwapFactory` contract
- The `StableSwapLP` contract
- The `StableSwapLPCollateral` contract
- The `StableSwapPool` contract
- The `StableSwapPoolInfo` contract
- The `Moolah` contract
- The `SmartProvider` contract

Consider adding a NatSpec comment containing a security contact above each contract definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Acknowledged, will resolve. The team stated:*

> *Will add in future versions*

N-17 Duplicate Import

The Moolah contract currently imports `ReentrancyGuardUpgradeable` twice. This redundancy can lead to confusion and maintainability issues.

Consider removing duplicate imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #76 at commit aae44ec.*

N-18 Missing Named Parameters in Mappings

Since Solidity 0.8.18, mappings can include named parameters to provide more clarity about their purpose. Named parameters allow mappings to be declared in the form `mapping(KeyType KeyName? => ValueType ValueName?)`. This feature enhances code readability and maintainability.

Throughout the codebase, multiple instances of mappings without named parameters were identified:

- The `stableSwapPairInfo` state variable in the `StableSwapFactory` contract.
- The `swapPairContract` state variable in the `StableSwapFactory` contract.
- The `position` state variable in the `Moolah` contract.
- The `isIrmEnabled` state variable in the `Moolah` contract.
- The `isLltvEnabled` state variable in the `Moolah` contract.
- The `isAuthorized` state variable in the `Moolah` contract.
- The `nonce` state variable in the `Moolah` contract.
- The `providers` state variable in the `Moolah` contract.

Consider adding named parameters to mappings in order to improve the readability and maintainability of the codebase.

Update: *Acknowledged, will resolve. The team stated:*

> *This upgrade will not include any modifications to the core contract unrelated to the smart collateral logic, in order to keep the scope minimal.*

N-19 Unused Imports

Within `StableSwapPoolInfo.sol`, multiple instances of unused imports were identified:

- The `SafeERC20` import
- The `StableSwapType` import

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #76 at commit 4e78821.*

Conclusion

The audited codebase introduces three core products for the ListaDAO ecosystem: a StableSwap AMM forked from PancakeSwap, the Moolah lending market forked from Morpho Blue, and a `SmartProvider` contract to integrate them.

The most severe finding is a reentrancy vulnerability in the `SmartProvider` contract that could allow an attacker to mint unbacked collateral, leading to a loss of funds for the protocol. Additionally, several medium-severity issues were identified, highlighting logical flaws in core mechanics such as liquidations, fee accrual, and EIP-712 signature validation. Other findings pointed to inconsistencies in enforcing protocol invariants, such as minimum loan sizes, and multiple issues related to access control that could lead to privilege escalation or loss of administrative capabilities.

While the system is built on battle-tested foundations from Morpho Blue and PancakeSwap, the recommendations of this audit focus on strengthening the custom modifications and integrations to ensure the protocol's robustness and security. Several fixes were suggested to improve the clarity of the codebase and facilitate future audits and development.

The ListaDAO team is appreciated for their collaboration and responsiveness during the audit process.