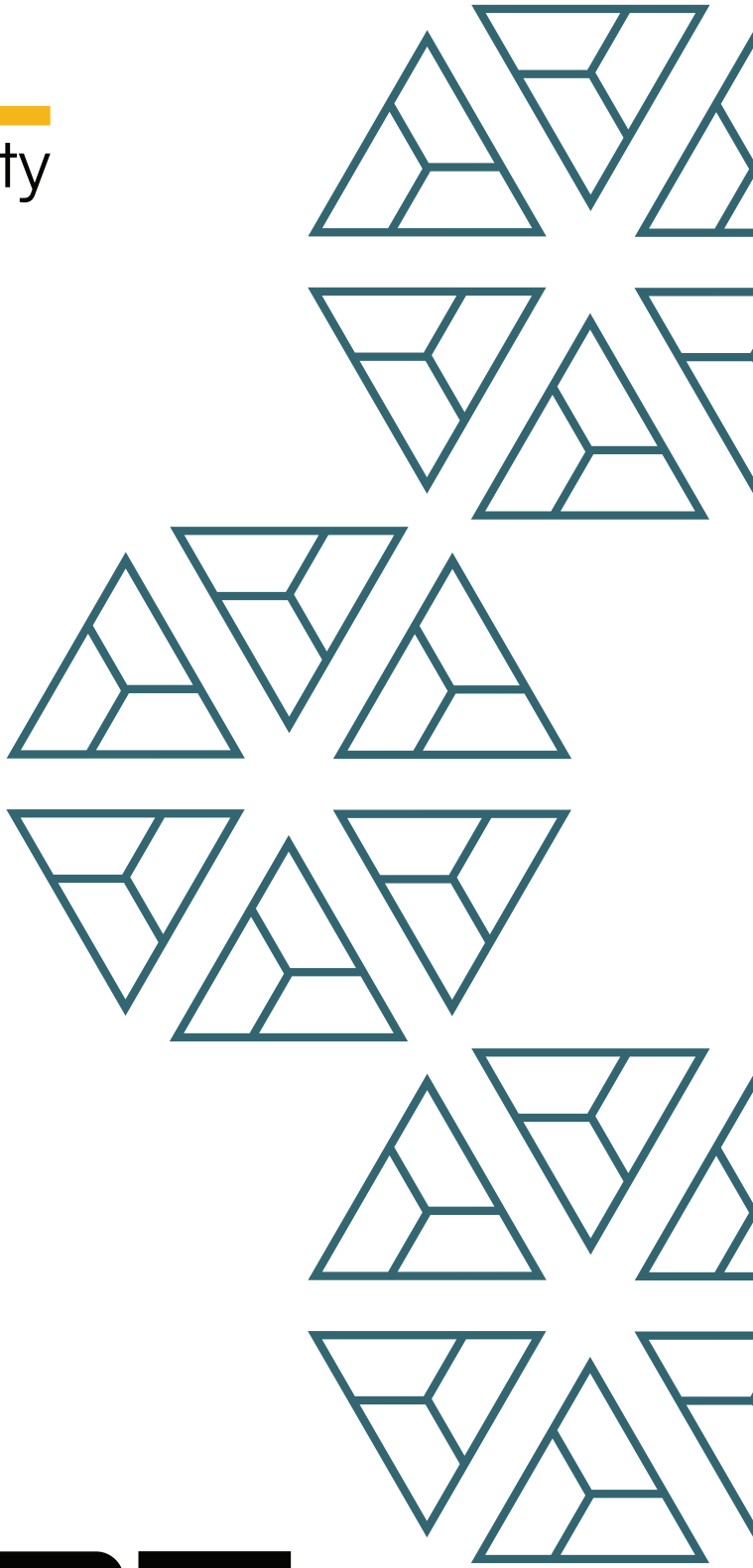

Lista DAO
Lista Lending

FINAL REPORT

March ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Lista DAO - Lista Lending
Website	lista.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/lista-dao/moolah/tree/423edc1dbb371de1f398d497815ec8757d49349b/src
Resolution 1	https://github.com/lista-dao/moolah/tree/23a8156b98a4a9c8adfadab5bbddd2db7ad295f3/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	8	2	2	3	1
Medium	16	1	2	11	
Low	6	1		5	
Informational	5	1		4	
Governance					
Total	35	5	4	23	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

3. Detection

Lista Lending

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

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

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

Appendix: Core Modifications

As a fork of Morpho Blue, Moolah introduces the following modifications:

1. The contract is now upgradeable using the UUPS pattern
2. Role management is handled via AccessControlEnumerableUpgradeable
3. All core functions are pausable by permissioned roles.
4. Admins can whitelist specific liquidators for certain markets.
5. All core functions are protected against reentrancy using the nonReentrant modifier.
6. A getPrice function has been added, which queries the oracle contract for a market and calculates the relative price between a collateral token and a loan token, factoring in the scaling coefficient.

Appendix: Permissionless Market Creation

The Moolah contract allows users to create markets using a whitelisted IRM and LLTV. These markets facilitate supplying collateral, borrow-lending operations, permissioned/permissionless liquidations and flash-loans.

Appendix: Supplying and Withdrawing

Users can supply collateral to any existing market by specifying either the amount of assets they want to deposit or the number of shares they want to receive. Upon supplying, users begin earning interest proportional to their share of the market.

Withdrawals work similarly: users can redeem either a specific amount or share of their deposit. All calculations are done using precise share-to-asset conversions, with rounding that favors the protocol for safety.

Appendix: Borrowing and Position Management

To open a borrow position, a user must first deposit collateral using the supplyCollateral function. This increases the user's collateral balance for a given market, uniquely identified by an Id derived from the provided MarketParams. Once sufficient collateral is deposited, the user can initiate a borrow by calling the borrow function, choosing to specify either the exact amount of assets they want to borrow or the number of borrowShares they want to receive. The contract handles the internal conversion between assets and shares based on current market totals. Also the borrower is required to create a healthy position.

When a borrow is executed, the user's borrowShares are increased, and loan assets are transferred to the specified receiver. These borrow shares represent the user's debt position and track their proportional responsibility in the total borrowed pool. Borrowed funds can be repaid at any time using the repay function, which accepts either asset or share input. Upon repayment, the user's borrow shares are reduced, and the repaid assets are returned to the protocol.

Collateral can also be withdrawn using the withdrawCollateral function, as long as the user's position remains healthy. The system continuously enforces health checks by comparing the user's collateral value (via oracle pricing and the market's lltv) against their outstanding borrow amount. If a withdrawal would leave the position undercollateralized, the transaction is reverted. This ensures all borrow positions are always properly backed by on-chain collateral, maintaining solvency and protecting lenders.

Appendix: Compounded Interest

Every time a user's position is modified—whether through borrowing, repaying, supplying, or withdrawing—interest is accrued on the total borrowed assets. This ensures the market remains up-to-date and fairly accounts for the cost of borrowing over time.

The interest is calculated using the formula:
$interest = totalBorrowedAssets * (e^{borrowRate*elapsed} - 1)$

Since computing the exponential function on-chain is expensive, the contract approximates it using a **Taylor series expansion** up to the third term:

$$e^{borrowRate*elapsed} =$$
$$= borrowRate * elapsed + (borrowRate * elapsed)^2 \div 2 + (borrowRate * elapsed)^3 \div 3$$

Appendix: Position Health

The position is healthy when the borrowed amount is less than the maxBorrow.
*maxBorrow = collateral * collateralPrice * lltv*
The **Loan-to-Liquidation Threshold Value** (lltv) defines the maximum portion of collateral value that can be borrowed against.
If the user's borrowed amount exceeds maxBorrow, their position is considered **unhealthy** and becomes eligible for liquidation.

Appendix: Flash Loans

The Moolah contract also supports **flash loans**, allowing users to borrow assets with zero upfront collateral, as long as the borrowed amount is returned within the same transaction. During the flash loan, Moolah transfers the requested assets to the caller, invokes the onMoolahFlashLoan function, and expects the exact same amount to be returned before the transaction ends. If the assets are not returned in full, the entire transaction reverts.

Appendix: Invariants

INV 1: Total borrowed assets must never exceed total supplied assets in a market.
INV 2: A user cannot create or modify a position in a way that would result in it becoming unhealthy.
INV 3: Shares must accurately reflect proportional ownership.
INV 4: Interest is accrued before every position-changing action to reflect real-time debt growth.
INV 5: Only governance-enabled IRMs and LLTVs can be used to create new markets.
INV 6: Flash loans must be repaid within the same transaction or the call reverts.

Privileged Functions

- enableIRM
- enableLltv
- setFee
- setFeeRecipient
- addLiquidationWhitelist
- removeLiquidationWhitelist

- pause
- unpause

Issue_01	Bad debt socialization can be skipped
Severity	**High**
Description	In the liquidate function the liquidator can specify the exact amount of collateral to seize from the position (seizedAssets) and the corresponding borrowed asset/shares is calculated. In the case of a bad debt, where the users collateral is not enough to fully cover their loan/borrowed assets, the remaining assets are removed from the total supplied assets, socializing the bad debt among all current lenders. The issue with this is that the bad debt is only socialized when the liquidator completely seizes all the borrowers collateral assets (i.e. *position[id][borrower].collateral == 0*) In a case where a liquidator is incentivized to prevent bad debt socialization (a case where liquidator also has supply shares in the market) or a liquidator simply acting in bad faith. Such liquidator can prevent the debt socialization by calling the liquidate function with a seizedAssets amount 1 wei less than the borrower's collateral, bypassing the bad debt check and possibly leading to insolvency issues.
Recommendations	A health check after liquidation can prevent this but might prevent partial liquidations. Another recommendation is to prevent partial liquidation if seizedAssets is above a threshold percent(say 90%) of the total collateral, forcing the liquidator to completely liquidate the position rather than liquidating just 90% or 99% e.t.c
Comments / Resolution	Partially resolved, If a borrower gets liquidated with an amount of remaining bad debt higher than the minimum loan amount, the liquidator can still pull this attack because _isHealthyAfterLiquidate will return true.

Issue_02	Unpausing the protocol will result in instant liquidations
Severity	**Medium**
Description	The Moolah contract is pausable by the PAUSER role. In case the protocol is paused for a long time, borrower's collateral can fall in price, or the position can become unhealthy due to interest accrual, which will result in positions becoming liquidatable without a way for borrowers to add collateral or repay their loans. When the protocol is unpaused by the MANAGER, instant liquidations will occur resulting in losses for the borrowers.
Recommendations	Consider adding a post-unpause grace period for liquidations to give time for borrowers to repay or add collateral to their positions.
Comments / Resolution	Acknowledged.

Issue_03	Min. supply assets can be bypassed
Severity	**Medium**
Description	The introduced _checkSupplyAssets function prevents users from having below minLoanValue worth of assets. However this check is not done after withdraw , which means users can withdraw any amount after supplying , possibly leaving less than minLoanValue worth of assets in their position.
Recommendations	Consider adding the _checkSupplyAssets check to the withdraw function as well
Comments / Resolution	

Issue_04	Flash Loan Attack via ERC-4626 Share Token Manipulation
Severity	**Medium**
Description	A known vulnerability in lending protocols allows attackers to manipulate the value of ERC-4626 share tokens using flash loans. In the context of the Moolah protocol, this can lead to bad debt if an ERC-4626 share token is used as collateral and most of its supply is deposited in Moolah. If exploited, this attack can artificially deflate the price of a collateral token, allowing an attacker to borrow more than they should and leave the protocol with unbacked debt. **Requirements:** • A Moolah market that accepts an ERC-4626 share token as collateral • Most (or all) of the share token supply is deposited in the Moolah market **Attack steps**: 1. The attacker acquires some of the ERC-4626 share token and uses it as collateral to borrow another asset from Moolah. 2. The attacker takes a flash loan of the ERC-4626 share token from the market, borrowing the entire token supply. 3. They redeem all the share tokens through the ERC-4626 vault, draining its underlying assets and resetting the share price. 4. They then mint new share tokens using fewer underlying assets (now that the share price is 1). 5. The attacker repays the flash loan with the newly minted shares. 6. The share price has now been reset, making the original collateral worth less than the borrowed amount, resulting in protocol bad debt. Moolah allows permissionless market creation, meaning anyone can create a market that accepts ERC-4626 tokens as collateral. If most of a token's supply ends up the protocol, the conditions for the exploit are met.
Recommendations	To mitigate this risk, it's advised to send to the dead address a

	small portion of both the collateral and loan assets during market creation. This ensures that not all of the token supply can be manipulated via the protocol.
Comments / Resolution	

Issue_05	Use of decimals function is not required in all ERC20s
Severity	**Low**
Description	The getPrice function calls the decimals function on the collateral and loan token to calculate the relative price between the two, adjusted by their decimals. However, according to the official EIP-20, the decimals function is optional, and contracts must not expect this function to be present on all ERC20s. Given that market creation on Moolah is permissionless, users that create markets must know of these limitations on Moolah to avoid the creation of broken markets.
Recommendations	Given that the decimals function is present in all popular ERC20 implementations, this issue can be safely acknowledged and documented so that markets are not created using tokens that do not implement the decimals function.
Comments / Resolution	Acknowledged.

Issue_06	Function getPrice only works when oracle prices have the same decimals.
Severity	**Low**
Description	The new getPrice function works as following: 1. It queries the decimals of the collateral and loan tokens 2. It queries the oracle price of both tokens 3. It calculates the scaled relative price between the two tokens by adjusting the oracle prices with a scaling factor. This process only works correctly when the oracle prices have the exact same decimals. While this is usually the case with Chainlink and Binance feeds, there are some price feeds that implement different decimals than the rest. Given that market creation is permissionless on Moolah, it's expected that some tokens are going to be used which have different decimals on their feeds. A market that uses these tokens will cause the getPrice function to be broken and cause a loss of funds to its users.
Recommendations	Given the permissionless nature of Moolah, it's safer to document on the code and docs the fact that a market can only work correctly when its oracle returns the prices in the same decimals for both tokens.
Comments / Resolution	Acknowledged.

Issue_07	Domain separator doesn't contain the contract version
Severity	**Low**
Description	As per EIP-712, signatures from different versions should not be compatible. The DomainSeparator for the Signatures on Moolah does not include the contract's version, and, given that Moolah is an upgradeable contract, the version may change.
Recommendations	Consider the contract version on the Domain Separator. This can also be acknowledged as its not a severe issue.
Comments / Resolution	Acknowledged.

Appendix: Known Morpho issues

Issue_08	Withdrawal or borrowing of loan tokens can be griefed
Severity	**High**
Description	With the current implementation of Morpho interest does not accrue within the same block and one is not disallowed to borrow and repay in the same block. These two properties open a griefing attack that: 1. Borrowing liquidity can be blocked. 2. Withdrawing liquidity by the suppliers can be blocked. To perform such attacks the attacker would only need to: 1. Pay for transaction gas fees. 2. Have enough collateral to be able to borrow all the remaining liquidity. The attack works as follows assume T1 is the transaction the honest actor wants to submit to either borrow or withdraw liquidity, then the attacker frontruns T1 with T0 which is the transaction to remove/borrow all the liquidity by providing enough collateral and back runs with T2 which is the transaction that would repay the loan and withdraws the collateral. Overall, the attacker would get all its collateral back but indeed would need to spend on gas and depending on the market state have enough liquidity to begin with. Griefing the borrow end point can be tolerable as the honest actor would still have access to their tokens. But blocking a supplier to withdraw their loan tokens is one that is more important.
Recommendations	To fix this issue, borrowing and repaying in the same block (or even a small time interval) should not be allowed to prevent this attack. However, since Morpho acknowledged this issue plus the fact that it is present in many different codebases, we recommend acknowledging it to inherit Morpho's robustness and prevent code changes.
Comments / Resolution	Acknowledged. Anyone can still borrow and repay in the same block to take all available liquidity and grief another borrower.

	Also, anyone can still withdraw all liquidity and supply it again atomically to grief another supplier or borrower.

Issue_09	Lack of sanity checks for createMarket
Severity	**High**
Description	The current implementation of createMarket is not covering all the marketParams attributes with sanity checks. This could allow the creation of broken markets or markets that do not make sense. Morpho should consider implementing the following sanity checks: • marketParams.loanToken != address(0) && loanToken != collateralToken. • marketParams.collateralToken != address(0) (the collateralToken != loanToken check is implicit because of the previous check). • marketParams.oracle.price() returns a valid answer and does not revert. • marketParams.irm is working as expected without reverting.
Recommendations	Consider implementing the suggested sanity check or clearly acknowledge them in the IMorpho natspec documentation, explaining which possible side effects could happen because those checks have not been performed.
Comments / Resolution	Partially resolved. Oracle.peek() could return 0 as price instead of reverting, but that is not a healthy oracle. Recommendation is to validate the returned price is > 0

Issue_10	First borrower of a Market can stop other users from borrowing by inflating totalBorrowShares
Severity	**High**
Description	Morpho tracks users' borrowing using shares and share prices similar to ERC4626. SharesMathLib.sol. When a user wants to borrow a certain amount of tokens (asset), Morpho contracts mint shares for the user using the formula: *assets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares)* Because the calculation rounds down borrowing shares, users can receive shares without incurring actual debt when totalBorrowAssets is equal to zero. Attackers can stop other users from borrowing by inflating totalBorrowShares without borrowing(by borrowing (1e6 - 1) shares). Once the totalBorrowShares is inflated, the value of assets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares) would easily exceed type(uint128).max with a normal value (e.g., 1 ether).
Recommendations	Consider adding a lower bound of borrowing assets for the first borrower or simply executing the first borrow first to prevent code changes and inherit Morpho's robustness.
Comments / Resolution	Resolved, now there's a minimum amount of debt that each borrower must hold.

Issue_11	Deviation in oracle price could lead to arbitrage in high lltv markets
Severity	**Medium**
Description	In Morpho Blue, the maximum amount a user can borrow is calculated with the conversion rate between loanToken and collateralToken returned by an oracle. However, all price oracles are susceptible to front-running as their prices tend to lag behind an asset's real-time price. More specifically: Chainlink oracles are updated after the change in price crosses a deviation threshold, (eg. 2.5% in ETH / USD), which means a price feed could return a value slightly smaller/larger than an asset's actual price under normal conditions. Uniwap V3 TWAP returns the average price over the past X number of blocks, which means it will always lag behind the real-time price. An attacker could exploit the difference between the price reported by an oracle and the asset's actual price to gain a profit by front-running the oracle's price update. For Morpho Blue, this becomes profitable when the price deviation is sufficiently large for an attacker to open positions that become bad debt. The likelihood of this condition becoming true is significantly increased when ChainlinkOracle.sol is used as the market's oracle with multiple Chainlink price feeds.
Recommendations	Consider implementing a borrowing fee to mitigate against arbitrage opportunities. Ideally, this fee would be larger than the oracle's maximum price deviation so that it is not possible to profit from arbitrage. Alternatively, this issue can be acknowledged with a notice on the frontend that no high lltv markets should be installed or additional validation during the creation of new markets.

Comments / Resolution	Acknowledged.

Issue_12	No functionality to disable IRM and LLTV
Severity	**Medium**
Description	The Morpho team has decided not to implement any functions for disabling the IRM and the LLTV which could cause issue if these turn out to be broken.
Recommendations	Consider adding functions to disable IRM and LLTV.
Comments / Resolution	Acknowledged.

Issue_13	Fees Can Be Avoided by Creating New Markets
Severity	**Medium**
Description	Morpho markets charge an interest rate to borrowers for their services, which is paid out evenly to all the liquidity providers who are supplying those tokens to be borrowed. Morpho has the ability to activate fees on a per-market basis, and those fees are taken out of the accrued interest. This fee collection mechanism effectively charges the liquidity providers exclusively, since whether this fee is charged is completely transparent to the borrowers. If liquidity providers see that a fee has just been configured for their market of choice, they are incentivized to create a brand new market, where the fee is by default set to zero, and migrate their liquidity to the new market as soon as it becomes available. As liquidity drains out, borrowers will be incentivized to borrow from the new market instead. Liquidity providers can create the same market over and over, with the exact same parameters. The reason they can do so is that the oracle address is arbitrary, so multiple different proxies can be created, all pointing to the same actual oracle implementation of choice. When doing so, the oracle address will be different (even though the implementation is the same), which will produce a unique hash for the Id .
Recommendations	Consider determining whether this is the intended behavior. Particularly, consider deciding whether it makes sense for identical markets to be able to exist with only different addresses for the oracle. Alternatively, consider restricting the functionality by having oracles being whitelisted, similar to what already happens to interest rate models.

Comments / Resolution	Resolved, now all new markets have a default fee of 5%.

Issue_14	Suppliers can be tricked into supplying more
Severity	**Medium**
Description	The supply and withdraw functions can increase the supply share price (totalSupplyAssets / totalSupplyShares). If a depositor uses the shares parameter in supply to specify how many assets they want to supply they can be tricked into supplying more assets than they want. It's easy to inflate the supply share price by 100x through a combination of a single supply of 100 assets and then withdrawing all shares without receiving any assets in return. The reason is that in withdraw we compute the assets to be received as *assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);.* Note that assets can be zero and the withdraw essentially becomes a pure burn function. Example: • A new market is created. • The victim tries to supply 1 assets at the initial share price of 1e-6 and specifies supply(shares=1e6). They have already given max approval to the contract because they already supplied the same asset to another market. • The attacker wants to borrow a lot of the loan token and therefore targets the victim. They frontrun the victim by supply(assets=100)

	and a sequence of withdraw() functions such that totalSupplyShares = 0 and totalSupplyAssets = 100. The new supply share price increased 100x. • The victim's transaction is minted and they use the new supply share price and mint 100x more tokens than intended (possible because of the max approval). • The attacker borrows all the assets. • The victim is temporarily locked out of that asset. They cannot withdraw again because of the liquidity crunch (it is borrowed by the attacker).
Recommendations	Suppliers should use the assets parameter instead of shares whenever possible. In the other cases where shares must be used, they need to make sure to only approve the max amount they want to spend. Alternatively, consider adding a slippage parameter maxAssets that is the max amount of assets that can be supplied and transferred from the user. This attack of inflating the supply share price is especially possible when there are only few shares minted, i.e., at market creation or when an attacker / contracts holds the majority of shares that can be redeemed.
Comments / Resolution	Partially resolved. Now there's a minimum amount check on supply, but that check should also be applied on withdraw. The attack can still be executed by supplying the minimum amount allowed, and repeatedly call withdraw to bring totalSupplyShares to zero.

Issue_15	Virtual supply shares steal interest
Severity	**Medium**
Description	The virtual supply shares, that are not owned by anyone, earn interest in _accrueInterest. This interest is stolen from the actual suppliers which leads to loss of interest funds for users. Note that while the initial share price of 1e6 might make it seem like the virtual shares can be ignored, one can increase the supply share price and the virtual shares will have a bigger claim on the total asset percentage.
Recommendations	The virtual shares should not earn interest as they don't correspond to any supplier. However, we recommend acknowledging this issue, because fixing it would introduce more complexity, potentially causing more issues.
Comments / Resolution	Acknowledged.

Issue_16	Virtual borrow shares accrue interest and lead to bad debt
Severity	**Medium**
Description	The virtual borrow shares, that are not owned by anyone, earn interest in _accrueInterest. This interest keeps compounding and cannot be repaid as the virtual borrow shares are not owned by anyone. As the withdrawable funds are computed as supplyAssets - borrowAssets, the borrow shares' assets equivalent leads to a reduction in withdrawable funds, basically bad debt. Note that while the initial borrow shares only account for 1 asset, this can be increased by increasing the share price. The share price can be inflated arbitrarily high.
Recommendations	There is no virtual collateral equivalent and therefore the virtual borrow assets are bad debt that cannot even be repaid and socialized. The virtual borrow shares should not earn interest. However, we recommend acknowledging this issue, because fixing it would introduce more complexity, potentially causing more issues.
Comments / Resolution	Acknowledged.

Issue_17	Any oracle update with sufficiently big price decline can be sandwiched to extract value from the protocol
Severity	**Medium**
Description	Whenever oracle update provides substantial enough price decline the fixed nature of liquidation incentive along with with the fixed LLTV for the market provides for the ability to artificially create bad debt and immediately liquidate it, stealing from protocol depositors. For highly volatile collateral it's possible to sandwich Oracle update transaction (tx2), creating min collateralized loan before (tx1) and liquidating it right after, withdrawing all the collateral (tx3). As liquidation pays the fixed incentive and socialize the resulting bad debt, if any, all the cases when bad debt can be created on exactly one Oracle update (i.e everywhere when it's sufficiently volatile asset respect to LLTV), this can be gamed, as the attacker will pocket the difference between debt and collateral valuation, which they receive in full via liquidate-withdraw collateral sequence in tx3. Notice that initial LLTV setting might be fine for the collateral volatility at that moment, but as particular collateral/asset pair might become substantially more volatile (due to any developments in collateral, asset or changes of the overall market state), while there is no way to prohibit using LLTV once enabled. Proof of Concept: • Morpho instance for stablecoins was launched with USDC collateral and USDT asset allowed, and with LLTV set included competitive reading of 95%, over time it gained TVL, USDC and USDT is now traded 1:1. Liquidation incentive factor for the market is liquidationIncentiveFactor = 1.0 / (1 - 0.3 * (1 - 0.95)) = 1.01522. • There was a shift in USDC reserves valuation approach, and updated reserves figures are priced in sharply via new Oracle reading of 0.9136 USDT per USDC. • Bob the attacker front runs the Oracle update transaction (tx2) with the borrowing of USDT 0.95m having provided USDC 1m (tx1, and for simplicity we ignore dust adjustments in the numbers where

	they might be needed as they don't affect the overall picture). • Bob back-runs tx2 with liquidation of the own loan (tx3), repaying USDT 0.9m of the USDT 0.95m debt. Since the price was updated, with repaidAssets = USDT 0.9m he will have seizedAssets = USDC 0.9m * 1.01522 / 0.9136 = USDC 1m. • Bob regained all the collateral and pocketed USDT 0.05m, which was written off the deposits as bad debt. So, an Attacker can steal principal funds from the protocol by artificially creating bad debt. This is a permanent loss for market lenders, its severity can be estimated as high. The probability of this can be estimated as medium as collateral volatility is not fixed in any way and sharp downside movements will happen from time to time in a substantial share of all the markets. The prerequisite is that initially chosen LLTV does not fully guarantee the absence of bad debt after one Oracle update. This effectively means that LLTV has to be updated to a lower value, but it is fixed within the market and such changes are usually subtle enough (as compared to more substantially scrutinized initial settings), so, once that happens, there is substantial probability that there will be a window of opportunity for attackers, the period when LLTV of a big enough market being outdated and too high, but there was no communication about that and the marker being actively used.
Recommendations	Consider introducing a liquidation penalty for the borrower going to fees. The goal here is not to increase the fees, it's recommended that interest based fee should be simultaneously lowered, but to disincentivize the careless borrowers and to make self-liquidation a negative sum game. Also, consider monitoring for the changes in collateral volatilities and flagging the markets whose initially chosen LLTV became not conservative enough, so lenders be aware of the risk evolution. Alternatively, consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_18	Users can take advantage of low liquidity markets to inflate the interest rate
Severity	**Medium**
Description	Morpho Blue is meant to work with stateful Interest Rate Models (IRM) whenever _accrueInterest() is called, it calls borrowRate() of the IRM contract. This will adjust the market's interest rate based on the current state of the market. For example, AdaptiveCurveIrm.sol adjusts the interest rate based on the market's current utilization rate. However, this stateful implementation will always call borrowRate() and adjust the interest rate, even when it should not. For instance, in AdaptiveCurveIrm.sol, an attacker can manipulate the market's utilization rate as such: • Create market with a legitimate loanToken, collateralToken, oracle and the IRM as AdaptiveCurveIrm.sol. • Call supply() to supply 1 wei of loanToken to the market. • Call supplyCollateral() to give himself some collateral. • Call borrow() to borrow the 1 wei of loanToken. • Now, the market's utilization rate is 100%. • Afterwards, if no one supplies any loanToken to the market for a long period of time, Adaptive CurveIrm.sol will aggressively increase the market's interest rate. This is problematic as Morpho Blue's interest compounds based on e^x. As such, when borrowRate (the interest rate) increases, interest will grow at an exponential rate, which could cause the market's totalSupplyAssets and totalBorrowAssets to become extremely huge. This creates a few issues: 1. The market will have a huge amount of unclearable bad debt: Should a large amount of interest accrue, totalBorrowAssets will be extremely large, even though totalBorrowShares is only 1e6 shares. Half of totalBorrowAssets would have actually accrued to the other 1e6 virtual shares. As such, after liquidating the attacker's 1e6 shares, half of

	totalBorrowAssets will still remain in the market as un-clearable bad debt. 2. The market will permanently have a high interest rate: As mentioned above, AdaptiveCurveIrm.sol aggressively increased the market's interest rate while there was only 1 wei supplied and borrowed in the market, causing utilization to be 100%. If other lenders decide to supply loanToken to the market, borrowers would still be discouraged from borrowing for an extended period of time as AdaptiveCurveIrm.sol would have to adjust the market's interest rate back down. 3. Users who call supply() with a small amount of assets might lose funds: If totalSupplyAssets is sufficiently large compared to totalSupplyShares, the market's shares to assets ratio will be huge. This will cause the following the share calculation in supply() to round down to 0: *if (assets > 0) shares = assets.toSharesDown(market[id].totalSupplyAssets, market[id].totalSupplyShares);* *else assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);* Should this occur, the user will receive 0 shares when depositing assets, resulting in a loss of funds.
Recommendations	In _accrueInterest(), consider checking that totalSupplyAssets is sufficiently large for IIrm.borrowRate() to be called. This prevents the IRM from adjusting the interest rate when the utilization rate is "falsely" high (e.g. only 1 wei supplied and borrowed, resulting in 100% utilization rate). Alternatively, consider acknowledging this issue.
Comments / Resolution	Partially resolved. The attack can still be executed with minLoanValue instead of 1 wei. If the attacker is the only user in a market, this exploit is unavoidable.

Issue_19	Users that borrow as much as possible (up to the LLTV threshold) will be liquidable in the very next block
Severity	**Low**
Description	Unlike protocols like Aave where they have both LTV (loan to value) and LT (liquidation threshold), the current implementation of Morpho offers just the LLTV parameter (Liquidation Loan-To-Value) without any buffer between the amount of debt that the user can take (given a collateral) and the liquidation threshold. This means that if a user borrows as much as it can borrow (given a collateral and LLTV), such user will be fully liquidable in the very next block as soon as the interest accrual can be triggered.
Recommendations	Morpho should carefully document this behavior and warn the users who perform such operation to be fully aware of the incumbent liquidation event they will incur on the very next block
Comments / Resolution	Acknowledged.

Issue_20	Replay Attacks Are Possible After Hardforks
Severity	**Low**
Description	If the event of a hard fork of the underlying chain that the Morpho contract is deployed to, the EIP712_DOMAIN_TYPEHASH value will become exploitable. This will happen because the chainId parameter is computed in the constructor. If a hard fork takes place after the contract's deployment, the chainId value used to compute the EIP-712 domain typehash will not be modified, so any signature used on the main chain will be replayable on the forked chain.
Recommendations	Consider checking the current chainId value every time against a cached value generated on the constructor in order to use the new one in case a hard fork is detected.
Comments / Resolution	Acknowledged.

MoolahVault

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

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

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

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

Appendix: Core Modifications

As a fork of Metamorpho, MoolahVault introduces the following modifications:

1. The contract is now upgradeable using the UUPS pattern
2. Role management is handled via AccessControlEnumerableUpgradeable
3. The timelock has been removed, as well as its related variables and functions.

Appendix: Deposit/Mint

Users deposit tokens into the vault (that in turn deposits to the underlying markets) by calling the deposit function.

On deposit the user provides the amount of assets to the vault, the vault transfers said assets from the user and then supplies them to the underlying markets in its supplyQueue. The tokens are supplied to markets in their order in the supplyQueue, if the tokens exceed the supplyCap of the current market, the leftover tokens are supplied to the next market in the queue.

All deposited tokens must be supplied otherwise the call reverts.

Mint follows the same process as deposit but in this case the user provides the desired amount of vault shares needed and the corresponding asset amount is deposited

Appendix: Withdraw/Redeem

On withdraw users provide the amount of assets to withdraw from the vault. The tokens are withdrawn from the underlying markets in the withdraw queue in order, if the amount of tokens to withdraw exceeds the maximum withdrawable amount for the current market the, the leftover is withdrawn from the next market in the queue.

On redeem the user provides the amount of vault shares to redeem and the corresponding amount of assets is withdrawn from the withdraw queue

Appendix: Reallocate

Re-distributes tokens between markets based on the provided allocations.
If the current supplyAssets in the market is greater than the allocation then the difference is withdrawn from the market and if it's less than the allocation then the difference is supplied to the market.
Ultimately the total token withdrawn from markets must equal the total token supplied to other markets or the reallocation fails.

Appendix: Invariants

INV 1: On reallocate total asset withdrawn must equal total asset supplied
INV 2: Only 0 supply cap market is removable from vault
INV 3: All user provided asset must be supplied to an underlying market
INV 4: Supply and withdraw queue cannot exceed the maximum queue length (30)

Privileged Functions

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

Issue_21	Inflation attack is possible if asset decimals is >/= 18
Severity	**Medium**
Description	The Inflation attack is possible on newly deployed vaults if the vault asset token decimals is >/= 18. The reason for this is because the DECIMAL_OFFSET becomes 0 in this case. The share<>asset conversion equation collapses to *share = asset * totalShares + 1/ totalAssets +1* *asset = shares * totalAssets +1/ totalShares +1* 1. An attacker deposit 1 wei of asset tokens on empty vault to receive 1 share 2. normal user sends tx to deposit 10,000 of asset tokens 3. attacker front-runs the user to supply 20,000 to the underlying market on behalf of the vault, inflating the vaults total assets to 20,000 + 1 = 20,001 4. user deposits but receives 0 shares *10,000 * (1+1)/(20,001 +1) = 20,000/20,002* which is rounded down to 0 In this case the attack is not profitable and the attacker can experiences losses, making it more of a griefing attack on the user
Recommendations	Consider ensuring that governance is always the first depositor, as even a revert on zero shares would not fully prevent such an attack because truncation to the downside can still happen, resulting in users receiving less shares than expected (e.g. 1 share instead of 1.9).
Comments / Resolution	Acknowledged.

Issue_22	Unused Guardian Role
Severity	**Informational**
Description	In the original version of the vault (Metamorpho), the guardian role was responsible for revoking time-locked actions such as removing markets or updating caps. In MoolahVault, the timelock mechanism has been removed, allowing governance to execute all actions immediately. As a result, the guardian role no longer serves a functional purpose.
Recommendations	Consider removing the guardian role from the contract to reduce unnecessary code and simplify the role structure.
Comments / Resolution	Acknowledged.

Issue_23	Missing Factory Contract for Vault Deployment
Severity	**Informational**
Description	When deploying a large number of vaults—such as in the case of MoolahVault—it is recommended to use a factory contract. A factory streamlines the deployment process and keeps track of deployed instances via a mapping. This approach not only simplifies deployments but also provides a reliable source of truth for users, helping them verify legitimate instances and avoid interacting with malicious or spoofed contracts.
Recommendations	Consider implementing a factory contract that deploys MoolahVault instances and records them in a mapping for easy reference and verification.
Comments / Resolution	Acknowledged.

Issue_24	Timelock removal allows Governance to execute actions instantly
Severity	**Informational**
Description	With the removal of the timelock, parameters such as supplyCap and withdrawQueue can now be modified instantly. This introduces a potential risk, as Governance could drastically alter the share price within a single transaction, without any delay or warning.
Recommendations	Timelock shall be added, so that users can react timely on such Governance changes.
Comments / Resolution	Acknowledged.

Issue_25	The vault will stop working if one of the Moolah market used by the vault stops working because of the IRM
Severity	**High**
Description	In a Morpho Blue market, the IRM component is called as soon as possible when one of the following operation is executed on the market itself: • withdraw • supply • borrow • repay • liquidate If the IRM breaks, each of those functions will revert, preventing those operations from being executed. In the MetaMorpho context, instead, the problem is amplified because one broken Morpho Blue market could break the whole Vault, preventing any users from withdrawing from the vault (even from the idle supply that is not exposed to any market directly). In particular, if one market is broken, all these functions will revert • reallocate if you specify a broken market in the withdrawn input • reallocate if you specify a broken market in the supplied input • deposit/mint, withdraw/redeem and setFeeRecipient if there's a broken market in the withdrawQueue (because internally, it will call totalAssets()) • submitFee/acceptFee, maxWithdraw/maxRedeem if there's a broken market in the withdrawQueue • convertToShares/convertToAssets (implemented by ERC4626) if there's a broken market in the withdrawQueue • previewDeposit/previewMint and previewWithdraw/previewRedeem (implemented by ERC4626) if there's a broken market in the withdrawQueue

	The big problem is that the broken market can't be removed from the withdrawQueue. To remove such a market, the allocator must call sortWithdrawQueue but the following conditions must be met to remove it: 1. The must be no supply shares on the market (otherwise it would mean that you are leaving funds over there) 2. The market cap must be equal to 0 (otherwise people could be able to supply but not withdraw from it) To reset the cap to zero, the curator must execute submitCap(brokenMarket, 0). This is not an issue because MORPHO is not called inside the logic and the function can't revert because of the broken market. The problem is when you want to remove the supply position that the vault itself has on the broken market. To accomplish that, you have to call reallocate and as we have already seen, that function will indeed revert because MORPHO.withdraw will revert.
Recommendations	Forcing the removal of a market from the withdrawQueue is an option to restore the Vault operationality, but it has huge side effects (without withdrawing first, all the supplier funds will be seen as "lost" and marked as removed). The force removal would also mean that one actor would have enough power to remove such market even if the market still have supply or a cap > 0. However, since it is a Morpho known issue, we recommend acknowledging it.
Comments / Resolution	Acknowledged.

Issue_26	Attackers can redistribute liquidity in MetaMorpho with flash loans and pose threats to smaller markets
Severity	**Medium**
Description	MetaMorpho deposits users' liquidity into the underlying morpho-blue markets whenever users deposit. In the _supplyMorpho function, metaMorpho iterates through all markets in the supplyQueue. If a market reaches its cap, metaMorpho deposits the liquidity into the next market until all the liquidity is distributed. Malicious users can exploit this feature to reallocate the liquidity within the metaMorpho. Let's assume there are 2 markets (A and B) in the supply queue with caps of 100M and 20M, respectively. Currently, there's 20M of metaMorpho's liquidity in market A and 1M in market B. Users can deposit 99M and withdraw it instantly to reallocate all the liquidity into market B'. Allocation of the liquidity would severely impact the performance of the metaMorpho. Allowing malicious users to game the portfolio would pose a huge challenge to the allocator. Another major concern would be the threats it poses to the underlying markets. As metaMorpho grows, liquidity becomes paramount for the underlying markets. If metaMorpho withdraws all liquidity from one market, the interest rates could be pushed to unusually high levels, endangering users of the underlying markets. **Note**: This issue is only possible when the withdraw and supply queue are in the same order
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_27	Virtual supply shares steal Interest
Severity	**Medium**
Description	The virtual supply shares that are not owned by anyone implicitly earn interest as the shares-to-asset conversions used in withdraw involve the virtual assets. *function _convertToSharesWithTotals(* *uint256 assets,* *uint256 newTotalSupply,* *uint256 newTotalAssets,* *Math.Rounding rounding) internal pure returns (uint256) { return assets.mulDiv(newTotalSupply + 10 ** _decimalsOffset(),* *newTotalAssets + 1, rounding);* *}* This interest is stolen from the actual suppliers which leads to loss of interest funds for users. Note that while the initial share price of 1e-6 might make it seem like the virtual shares can be ignored, one can increase the supply share price and the virtual shares will have a bigger claim on the total asset percentage
Recommendations	The virtual shares should not earn interest as they don't correspond to any supplier However we recommend acknowledging this issue because fixing it would introduce more complexity, potentially causing more issues
Comments / Resolution	Acknowledged.

Issue_28	Allocator can drain the MetaMorpho vault if a future IRM queries token balance
Severity	**Medium**
Description	The Allocator role in the MetaMorpho vault is responsible for distributing the portfolio and managing the risks of the vault. The Allocator can not supply the assets to an unauthorized market (market with supplyCap == 0). Thus, in the scenario where the allocator's key is breached,the vault should have limited damage. The issue lies at MetaMorpho.sol#L399: *if (suppliedAssets == 0) continue;* When an unauthorized market with suppliedAsset == 0 is provided in reallocate , the function just skips instead of reverting. This creates an attack vector. Since MetaMorpho calls morpho.accrueInterest in the loop, the malicious allocator could potentially get the control flow within reallocate function. Currently, only AdaptiveCurveIrm can be used, which wouldn't give users control flow and thus prevent the potential attacks. However, let's consider two hypothetical scenarios: 1. A new IRM is deployed that queries token balances to calculate interest. (This is a reasonable setting, as a lot of IRM actually depends on token's balance.) 2. Morpho-blue allows users to permissionlessly deploy their own IRMs. Given this assumption, the malicious allocator can do the following: 1. Deploy a fake market with a malicious callback function. 2. Triggers reallocate with three Allocations . The first and the third Allocation are enabled markets but in the second Allocation , the fake market is provided. 3. MetaMorpho processes each allocation: 1. The first Withdrawal is a correct one and the vault pulls tokens from the first market. The second market is malicious. 2. The attacker get the control flow through the IRM

	3. In the malicious callback, the exploiter deposits to MetaMorpho. Since the tokens had been pulled in the previous step, the vault's price is lower. The exploiter gets vault's shares at a low price. 4. Withdraw from the metaMorpho and get the profit
Recommendations	Consider reverting on reallocate if market is not enabled before calling _accruedSupplyBalance due to the potential reentrancy risk. Alternatively, this can be acknowledged.
Comments / Resolution	Acknowledged.

Liquidator

The Liquidator contract plays a key role in Moolah's risk management by handling liquidations of undercollateralized positions. It supports upgrades via UUPS and uses role-based access control with three main roles: admin, manager, and bot.

There are two types of liquidation: standard and flash. Both allow the contract to seize collateral and repay debt when a borrower's position becomes risky. For flash liquidations, the contract can swap seized assets for loan tokens using external DEXs. Only addresses with the BOT role can trigger these actions, and a callback handles post-liquidation steps.

The contract includes slippage protection, ensures borrowed assets are properly repaid, and has tools for managing tokens—like withdrawing assets, managing a whitelist of tradeable tokens, and enforcing minimum output on swaps. All actions are gated by roles to keep things secure.

Appendix: Core Modifications

As a fork of Morpho's Liquidator contract, this Liquidator introduces the following modifications:

1. The contract is now upgradeable using the UUPS pattern
2. Role management is handled via AccessControlEnumerableUpgradeable
3. Liquidations can happen with or without swaps in between, via the functions liquidate and flashLiquidate.
4. The contract now can hold funds, so that it can execute liquidations without swaps.
5. There's a token whitelist so that the BOT can swap tokens via sellToken.
6. There are new permissioned functions such as setTokenWhitelist, withdrawERC20 and withdrawETH.

Appendix: Flash and Normal Liquidations

In a **normal liquidation**(liquidate()) function is called with the target marketId, borrower address, amount of collateral (seizedAssets) to seize, and a swap pair address. The Liquidator retrieves the market parameters from the Moolah contract using idToMarketParams, and calls Moolah.liquidate(), passing the collateral and loan token, along with encoding a MoolahLiquidateData struct. The struct includes swap-related information, but the swap flag is set to false, indicating that no internal token swap will occur during the callback(onMoolahLiquidate()).

Flash liquidation(flashLiquidate()) performs the same logic but sets the swap flag to true and includes the encoded swapData to execute a token swap after seizing the collateral. The

flashLiquidate function works similarly to liquidate, but upon receiving the callback from Moolah through onMoolahLiquidate(), the contract performs a token swap by calling the external swap pair contract with the provided swapData. The seized collateral (collateralToken) is approved for the pair, and the contract expects to receive at least the repaidAssets amount in loanToken. If the output is insufficient to cover the repaid amount, the transaction reverts with NoProfit().

Both liquidate and flashLiquidate are permissioned functions that can only be used by the BOT role.

Appendix: Token Management

The Liquidator contract has the following functions that manage ERC20 tokens:

- withdrawERC20 - This function can only be called by the MANAGER role and transfers inputted token and amount to the manager.
- withdrawETH - Permissioned function that can only be called by the MANAGER to withdraw a specified amount of ETH.
- sellToken - Permissioned function that can only be called by the BOT to sell tokenIn for tokenOut

Appendix: Invariants

INV 1: Only BOT role can call liquidate, flashLiquidate, and sellToken.
INV 2: Only whitelisted tokens can be used in swaps.
INV 3: Contract must not use more than the amountIn during swaps; output must meet amountOutMin.
INV 4: If a swap is used in liquidation(flashLiquidate), it must return at least repaidAssets, or revert.
INV 5: Only MANAGER can withdraw ETH or ERC20 tokens from the contract.
INV 6: onMoolahLiquidate can only be called by the immutable MOOLAH address.

Privileged Functions

- flashLiquidate
- liquidate
- withdrawERC20
- withdrawETH
- setTokenWhitelist
- sellToken

Issue_29	BOT Can Self-Sandwich to Extract Funds During sellToken and flashLiquidate Swaps
Severity	**High**
Description	The BOT can manipulate its own transactions when calling sellToken or flashLiquidate to sandwich the swap and extract value, effectively stealing funds from the contract. **Example: Exploiting sellToken** The BOT can perform the following sequence in a transaction bundle: 1. Imbalance a DEX pool by executing a large swap using a flash loan. 2. Call sellToken, setting amountOutMin = 0 and selecting the now-imbalanced DEX pool. a. Due to the prior manipulation, the sellToken swap suffers significant slippage, resulting in a much lower output than expected. 3. Rebalance the DEX pool and repay the flash loan, allowing the BOT to pocket the profit created by the artificially induced slippage. This sequence effectively steals value from the contract by routing the slippage loss through sellToken, with the profit captured by the BOT. A similar attack path exists with flashLiquidate, where the BOT sandwiches the internal swap performed in onMoolahLiquidate. By causing slippage during the liquidation swap, the BOT can steal the value that is not needed to repay the loan, i.e. the liquidation incentive.
Recommendations	It's recommended to implement slippage checks in sellToken and flashLiquidate by trusted roles to ensure excessive slippage cannot occur. Alternatively, it's recommended to have a completely trusted role operating as the BOT.

Comments / Resolution	Acknowledged, the protocol has stated that they will closely monitor the BOT to ensure no exploit happens.

Issue_30	BOT Can Steal Funds from Liquidator via Fake Moolah Markets
Severity	**High**
Description	The BOT can create custom markets on Moolah to exploit the Liquidator contract and steal its funds. This attack can be executed atomically in a single transaction. **Attack overview:** 1. The BOT observes that the Liquidator contract holds 100 USDC. 2. It creates a new Moolah market using: a. **USDC** as the loan token b. A **custom (worthless) ERC20** as the collateral token 3. The BOT supplies 100 USDC to this new market. 4. It then deposits some of the fake collateral and borrows 100 USDC. 5. The BOT manipulates the custom oracle to make the loan eligible for liquidation. 6. It calls liquidate on the Liquidator contract, targeting the fake market. 7. The contract uses its own 100 USDC to repay the loan, and seizes the fake collateral in return. 8. Now, the BOT holds: a. 100 USDC borrowed from Moolah b. 100 USDC stolen from the Liquidator Effectively, the BOT ends up with **200 USDC**, and the Liquidator is left holding worthless collateral. This attack can be repeated as long as funds are available in the Liquidator contract. Since the entire process is atomic, the BOT faces no risk of loss.

Recommendations	It is recommended to implement checks to ensure only approved tokens and markets are used in liquidations: - The liquidation callback should verify both tokens are included in tokenWhitelist. - Alternatively, maintain a dedicated whitelist of approved Moolah markets. Alternatively, it's recommended to have a completely trusted role operating as the BOT.
Comments / Resolution	Resolved, now there's a market whitelist controlled by the governance.

Issue_31	BOT can pass arbitrary addresses and data in the external call
Severity	**High**
Description	In the sellToken function, the bot will specify the pair address and the swapData. This can be exploited in the following ways: 1. Malicious BOT can call any of the ERC20 approve functions: - Attacker can simply pass any ERC20 token address with approve data. By doing this they can approve any address to spend on behalf of the Liquidator contract 2. Reentrancy: - Malicious BOT can pass a malicious pair address, reentering the sellToken function to steal funds. 3. Passing any address as the recipient of the swap funds - The 1inch swap takes the recipient as a parameter. BOT can pass any address as the recipient and directly steal funds, by also specifying amoutOutMin = 0 .
Recommendations	The pair contract shall be set to the 1inch router. Also amountOutMin and the recipient field of the swapData shall be validated. Alternatively, it's recommended to have a completely trusted role operating as the BOT.

Comments / Resolution	Not resolved The calldata is still not checked, which allows the BOT to steal funds by setting a custom recipient that is not the Liquidator contract. It is recommended to ensure that the BOT role is fully trusted.

Issue_32	Approval is not reset to 0 after a swap is performed
Severity	**Low**
Description	The onMoolahLiquidate and the sellToken functions will perform a swap using the 1inch router: *SafeTransferLib.safeApprove(tokenIn, pair, amountIn);* *(bool success,) = pair.call(swapData);* However the approval is not reset back to 0 after the swap is performed. This may cause issue as some ERC20 tokens may have approval race protection.
Recommendations	Consider resetting the approval to 0 after the swap is performed.
Comments / Resolution	Resolved, now the approval is reset after the swap.

VaultAllocator

The VaultAllocator serves as a publicly accessible allocator for MoolahVault vaults. Its primary function is to enable capital reallocation across different markets by withdrawing assets from one or more markets and supplying them to a target market, all while enforcing configurable flow caps that limit how much can be moved in and out of each market. The contract implements role-based access control, allowing vault owners and designated admins to configure parameters like flow caps, admin addresses, and transaction fees that must be paid when reallocating funds.

At its core, the contract facilitates efficient capital management for vault owners through the reallocateTo function, which anyone can call provided they pay the required fee. This design enables vault owners to outsource their capital reallocation strategies while maintaining control over the parameters and limits of these reallocations.

Appendix: Core Modifications

As a fork of Morpho's PublicAllocator, VaultAllocator introduces the following modifications:

1. The contract is now upgradeable using the UUPS pattern
2. Role management is handled via AccessControlEnumerableUpgradeable

Appendix: ReallocateTo

The main functionality of the VaultAllocator is to allow users to reallocate tokens between the underlying markets within a vault. This reallocation process is controlled by the specified vault fee, and movements of tokens are limited by the configured flow caps set for each individual market. Once the new market allocations have been determined, the vault is called with the updated allocation data for execution.

Appendix: Invariants

INV 1: Only vault admin or manager can set market flowCaps
INV 2: Only vault admin or manager can claim accrued vault fees
INV 3: On reallocateTo, total withdrawn assets must be supplied to supply market and cannot exceed maxIn of the supply market
INV 4: Vault assets can only be reallocated on enabled markets

Privileged Functions

- setAdmin
- setFee

- setFlowCaps
- transferFee

Issue_33	Unused MANAGER role
Severity	**Informational**
Description	In the VaultAllocator::initialize function the MANAGER role is granted. However, does not have access to any of the permissioned functions. This is because the onlyAdminOrVaultOwner modifier will check if the msg.sender has the MANAGER role for the MoolahVault, but not for the VaultAllocator: *modifier onlyAdminOrVaultOwner(address vault) { if (msg.sender != admin[vault] && !IMoolahVault(vault).hasRole(MANAGER, msg.sender)) { revert ErrorsLib.NotAdminNorVaultOwner(); }*
Recommendations	Consider deleting this role.
Comments / Resolution	Resolved, the MANAGER role has been removed.

Issue_34	transferFee() uses transfer() which could cause problems if fee recipient has to execute logic on receive
Severity	**Informational**
Description	The transfer functions has a built-in gas limit of 2300 units. This will cause reverts of the transferFee function when the feeRecipient implements any logic, that is triggered upon receive.
Recommendations	If feeRecipient is meant to perform any operations on receive, consider using a low level call instead.
Comments / Resolution	Acknowledged.

Issue_35	Funds can be redirected to the idle market by reaching the MetaMorpho supply cap using a flash loan
Severity	**Medium**
Description	The public allocator allows the users to move the funds in the MM permissionless. However, to prevent undesired states in the MM portfolio, FlowCaps are configured, placing limitations on users' capabilities. One feature of the publicAllocator is to permit users to revert to flow caps in case the portfolio management is suboptimal. Reversing the publicAllocator action does not equate to reversing MM portfolio management, as the portfolios of MM may undergo changes between two publicAllocator actions. Consider the following configuration of MM and the public allocator: **supplyQueue** **market** **assets** **cap** 0 stETH 1,000,000 5,000,000 1 idle market 0 unlimited **withdrawQueue** **market** **assets** **cap** 0 idle market 0 unlimited 1 stETH 1,000,000 5,000,000 2 rETH 1,000,000 5,000,000 **publicAllocator** **market** **maxIn** **maxOut** 0 idle market 0 10,000,000 1 stETH 1,000,000 0 2 rETH 1,000,000 0 The malicious users can move funds into the idle market and place the MM in an undesired state with the following steps: 1. Flashloan and calls metaMorph.deposit with 5,000,000 assets. This fills up the first market (stETH market) and an extra 1,000,000 goes to the idle market. The users get 5,000,000 worth of MM shares in this step. **states** **supplyQueue** **market** **assets** **cap**

	0	stETH	5,000,000	5,000,000
	1	idle market	1,000,000	unlimited
	withdrawQueue	**market**	**assets**	**cap**
	0	idle market	1,000,000	unlimited
	1	stETH	5,000,000	5,000,000
	2	rETH	1,000,000	5,000,000
	publicAllocator	**market**	**maxIn**	**maxOut**
	0	idle market	0	10,000,000
	1	stETH	1,000,000	0
	2	rETH	1,000,000	0

2. Let PublicAllocator withdraw assets from the idle market by calling publicAllocator.reallocateTo, with the the idle market as the only withdrawal and the rETH market as the supplying market:

states:

supplyQueue	**market**	**assets**	**cap**
0	stETH	5,000,000	5,000,000
1	idle market	0	unlimited
withdrawQueue	**market**	**assets**	**cap**
0	idle market	0	unlimited
1	stETH	5,000,000	5,000,000
2	rETH	2,000,000	5,000,000
publicAllocator	**market**	**maxIn**	**maxOut**
0	idle market	1,000,000	9,000,000
1	stETH	1,000,000	0
2	rETH	0	1,000,000

3. Withdraw all the MM shares that the exploiter got in the first step. The MM would try to pull 5,000,000 worth of assets. Since there are no assets in the idle markets, MM pulls assets from the second markets in the withdrawal queue. MM pulls 5,000,000 asses from the stETH market.

states:

supplyQueue	**market**	**assets**	**cap**
0	stETH	0	5,000,000
1	idle market	0	unlimited
withdrawQueue	**market**	**assets**	**cap**

	0 idle market 0 unlimited 1 stETH 0 5,000,000 2 rETH 2,000,000 5,000,000 **publicAllocator market maxIn maxOut** 0 idle market 1,000,000 9,000,000 1 stETH 1,000,000 0 2 rETH 0 1,000,000 4. Let publicAllocator reverse the previous cap flow and move funds out of the idle market **states:** **supplyQueue market assets cap** 0 stETH 0 5,000,000 1 idle market 1,000,000 unlimited **withdrawQueue market assets cap** 0 idle market 1,000,000 unlimited 1 stETH 0 5,000,000 2 rETH 1,000,000 5,000,000 **publicAllocator market maxIn maxOut** 0 idle market 0 10,000,000 1 stETH 1,000,000 0 2 rETH 1,000,000 0 In the above example, the flowCaps remain the same after funds are moved to the idle market. Thus, the exploit still works even with a lower non-zero flowCaps limit. For instance, assume the maxOut of the idle market is 500,000, the exploit steps would be: 1. Deposit 4,500,000 in the first step, letting MM deposit 500,000 into the idle market. 2. Have the public allocator move 500,000 from the idle market to the rEth market. 3. Withdraw all shares from MM. At this point, 500,000 assets still exist in the stETH market. 4. Reverse the cap flow and move funds from rETH to the idle market. 5. Repeat steps 1 to 4.
Recommendations	Consider imposing more constraints on the publicAllocator, and here are some potential paths:

	1. Avoid allowing the reverse of cap flow. Do not increase maxIn when the publicAllocator withdraws from one market, and do not increase maxOut when the publicAllocator supplies to one market. 2. Set maxIn and maxOut of most markets to 0. For example, in the ETH market, we can mitigate the exploit by setting the maxIn of the rEth market to 0. 3. Consider charging a higher fee. It's important to note that the exploit can still be profitable even with fees turned on. To nullify the attack vector, fees should be charged in proportion to the funds being moved. Each of the above solutions comes with trade-offs, and there is no easy solution. We recommend paying close attention to new implementations. Alternatively, this can be acknowledged.
Comments / Resolution	Acknowledged.

InterestRateModel

The InterestRateModel contract implements a dynamic interest rate mechanism for the Moolah lending protocol. It adjusts borrowing rates based on market utilization, using a mathematical model that aims to maintain a target utilization rate. When utilization exceeds or falls below the target, the contract gradually adjusts the interest rate to incentivize market balance, with rates increasing as utilization rises above target and decreasing when utilization falls below target.

The contract employs a sophisticated approach to rate calculation that accounts for time elapsed since the last update, featuring exponential adaptation of rates and trapezoidal approximation for determining average rates. This design allows for responsive yet stable interest rate adjustments that help maintain optimal capital efficiency while incorporating safeguards like minimum and maximum rate boundaries to prevent extreme fluctuations.

Appendix: Core Modifications

As a fork of Morpho's AdaptativeCurveIRM, InterestRateModel introduces the following modifications:

1. The contract is now upgradeable using the UUPS pattern
2. Role management is handled via AccessControlEnumerableUpgradeable

Appendix: The Borrow Rate

The borrow rate is calculated using the adaptive curve interest rate model. Unlike traditional interest rate models with a fixed rate at target utilization, lower rates below the target utilization and higher rates above it.
The adaptive curve IRM has a dynamic rate at the target utilization (90%), this means the rate at utilization, $r_{90\%}$,changes with time depending on the current utilization.
With an initial $r_{90\%} = 4\%$, when utilization is below target (90%) the rate at target utilization ($r_{90\%}$) decreases with each subsequent rate calculation lowering the rate curve leading to an overall decrease in rates over time.
Similarly, when the current utilization is above the target, $r_{90\%}$ increases with each subsequent rate calculation raising the rate curve and leading to even higher rates over time.

Appendix: Invariants

INV 1: Rate at target Utilization cannot fall below 0.1% or exceed 200%
INV 2: Borrow rate can only be queried by the Moolah contract

No Issues Found