Cyberscope

Audit Report

Hakura Protocol

February 2024

Repository https://github.com/Hakura-io/HakuraProtocol

Commit 386483973c21ad4cc9791f85825d5dac0b1f00bd

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/Hakura-io/HakuraProtocol
Commit	386483973c21ad4cc9791f85825d5dac0b1f00bd

Audit Updates

Initial Audit	22 Feb 2024

Source Files

Aave-v2 Folder

Filename	SHA256
PositionsManagerUtils.sol	b29e73a6bcdbb1e13357a8175d63de9af4a3a63f84a6cc877c838df4c3e5a916
MorphoUtils.sol	c1495a667d63d23b80c9281b0695b0d07e3e97d9d018ac1290fbfd8e8f33deb8
MorphoStorage.sol	072e724c0c73ec49851f8cbe938d35e8d2a97056ca6b1a33be2050751bfe17d8
MorphoGovernance.sol	860bbb2a122639ca47014298d0dc7597a068ebf64e6cfa79c2d4a9b097967649
Morpho.sol	7bc4767cc5e06ff7d7f3b1a226d072f99dbab1653266f46b70409e0784c3950e
MatchingEngine.sol	de2de6b580a7619556777d737626378f66225918b833abc282231e6ba121b931
InterestRatesManager.sol	6b79d56babb93d12b4236f1f6893b5b05c954bd1727ba6c9bb9131d39a20ea08

ExitPositionsManager.sol	8d012b2be0dca0fe84dbed7ffd2e4c6911583e282dc682a30b3be98aeb939c10
EntryPositionsManager.sol	ca198568dcc6fe42956f374567e7f4d4d1c58031ca333cc6526331e7ba6e79b2

Common Folder

Filename	SHA256
RewardsDistributor.sol	ddf149e290b016606531823f271c25237ac3ba9ba247587fd3b92d1f87c5c8bc

Compound Folder

Filename	SHA256
RewardsManager.sol	0972b317dc9fd6474b56e7861358c39fed2d38007a06c1643caf3a873c4b1d28
PositionsManager.sol	68cadb585e3a52a93109ce61b41db884014aea93488a28f0dbb58ada756ba60d
MorphoUtils.sol	ccb81b10514fddf53892215c2a2170b10679c4e65de5ad1531fc05fce15c096f
MorphoStorage.sol	3fded8a2331606ffd44155423ecf5645fbf7a8d9b136f10172edf6b419cf25e4
MorphoGovernance.sol	0a87a079a1a4f6bbd3bb7c631ae83329dcf8d43683e962d78d839ca35f408d51
Morpho.sol	b96e22f593837df70faf140532138c4c6466d8e7c861f773c2950e19d29c2a3e
MatchingEngine.sol	c2b0511b5840eee1654727d211cce428830553c7caa5d3c71058e21fc244ec45

InterestRatesManager.sol	71aad03983d7041b766a4a46084fd7c493889dda17ae398154b10cae332e70a3

Overview

Morpho

Morpho is the main contract of the protocol and the entrypoint through which users supply and borrow liquidity. This contract inherits from the MorphoGovernance, MorphoUtils, and MorphoStorage contracts, among others, which all define helper and administrative functions to manage the liquidity pool. We reviewed Morpho and the contracts it inherits from to ensure the soundness of the system's architecture and to detect issues related to the use of delegatecall.

PositionsManager

This contract inherits from the MatchingEngine and MorphoStorage contracts, which provide the core functionality for managing users' deposits, withdrawals, borrowing operations, and repayments. The PositionsManager contract acts as a library that the main Morpho entrypoint contract calls (via delegatecall). We reviewed PositionsManager to ensure that it performs accurate bookkeeping and properly liquidates users at the Morpho level. We also searched for edge cases that could cause the underlying Morpho account in Compound to be liquidated.

InterestRatesManager

This contract contains the arithmetic required to calculate accrued interest. Like PositionsManager, it inherits from MorphoStorage and acts as a library that the main Morpho entrypoint contract calls (via delegatecall). We reviewed InterestRatesManager to ensure that it performs accurate bookkeeping for peer-to-peer (P2P) rates.

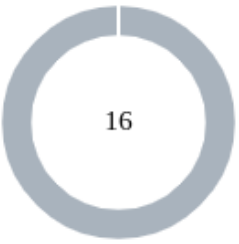
RewardsManager

This contract contains the logic required to manage and claim liquidity mining rewards provided by the underlying Compound protocol. It manages its own local storage and is called by the main Morpho contract to pass rewards to users. We reviewed this contract to ensure that it accurately tracks users' rewards and stays in sync with Compound.

IncentivesVault

This contract contains the logic required to manage protocol fees collected by the Morpho DAO. It manages its own local storage and is called by the main Morpho contract. It also gives users the option to claim rewards in Morpho tokens rather than COMP tokens. We reviewed IncentivesVault to ensure that it properly calculates incentive amounts for users.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CCR	Contract Centralization Risk	Unresolved
	DDP	Decimal Division Precision	Unresolved
	DLF	Deprecation Logic Flaw	Unresolved
	DLF	Desynchronization Liquidation Flaw	Unresolved
	MOPV	Missing Oracle Price Validation	Unresolved
	MZRC	Missing Zero Repayment Check	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	RSW	Redundant Storage Writes	Unresolved
	RTO	Rewards Transition Oversight	Unresolved
	USF	Unbounded Sorting Functionality	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L14	Uninitialized Variables in Local Scope	Unresolved
	L16	Validate Variable Setters	Unresolved

●	L19	Stable Compiler Version	Unresolved
●	L22	Potential Locked Ether	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	compound/MorphoGovernance.sol#L189,208,215,260,272
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

The owner of MorphoGovernance contract has excessive powers. Inter alia, the owner can pause critical functionalities and change the `positionsManager`, `rewardsManager`, and `treasuryVault` addresses, which allows the owner to gain control over users' funds. In the current implementation, the system depends heavily on the owner of the contract. Thus, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if the owner's private keys become compromised.

```
    function setPositionsManager(IPositionsManager _positionsManager)
external onlyOwner {
        if (address(_positionsManager) == address(0)) revert
ZeroAddress();
        positionsManager = _positionsManager;
        emit PositionsManagerSet(address(_positionsManager));
    }

    function setRewardsManager(IRewardsManager _rewardsManager) external
onlyOwner {
        rewardsManager = _rewardsManager;
        emit RewardsManagerSet(address(_rewardsManager));
    }

    function setTreasuryVault(address _treasuryVault) external onlyOwner
{
        treasuryVault = _treasuryVault;
        emit TreasuryVaultSet(_treasuryVault);
    }
    function setIsSupplyPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isSupplyPaused = _isPaused;
        emit IsSupplyPausedSet(_poolToken, _isPaused);
    }

    function setIsBorrowPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        if (!_isPaused && marketPauseStatus[_poolToken].isDeprecated)
revert MarketIsDeprecated();
        marketPauseStatus[_poolToken].isBorrowPaused = _isPaused;
        emit IsBorrowPausedSet(_poolToken, _isPaused);
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's

self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	compound/PositionsManager.sol#L361
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
delta.p2pSupplyDelta -= remainingToBorrow.div(poolSupplyIndex);
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

DLF - Deprecation Logic Flaw

Criticality	Minor / Informative
Location	aave-v2/MorphoGovernance.sol#L318,358 compound/MorphoGovernance.sol#L321,368
Status	Unresolved

Description

The contract is designed to manage the lifecycle of markets, including their deprecation. A critical part of this management involves ensuring that markets are not deprecated under conditions that could adversely affect the platform's operations or user interactions. Specifically, the current implementation requires that borrowing be paused before a market can be deprecated, as indicated by the `setIsDeprecated` function. However, there is a notable oversight in the liquidation process. The contract lacks a corresponding check in the `setIsLiquidateBorrowPaused` function to prevent the deprecation of a market if liquidation of borrowings is paused. This omission means that if the `isLiquidateBorrowPaused` flag is set to true, indicating that liquidations are paused, a market can still be deprecated. This scenario could lead to a situation where liquidators are unable to liquidate borrowers in a deprecated market, potentially freezing the market in an undesirable state and impacting the platform's liquidity and risk management.

```
    function setIsLiquidateBorrowPaused(address _poolToken, bool
_isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isLiquidateBorrowPaused =
_isPaused;
        emit IsLiquidateBorrowPausedSet(_poolToken, _isPaused);
    }

    function setIsDeprecated(address _poolToken, bool _isDeprecated)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        if (!marketPauseStatus[_poolToken].isBorrowPaused) revert
BorrowNotPaused();
        marketPauseStatus[_poolToken].isDeprecated = _isDeprecated;
        emit IsDeprecatedSet(_poolToken, _isDeprecated);
    }
```

Recommendation

It is recommended to modify the `setIsDeprecated` function to include a check that ensures the `isLiquidateBorrowPaused` flag is false before allowing the deprecation of a market. This ensures that markets can only be deprecated when both borrowing and liquidation activities are not paused, maintaining the platform's operational integrity. Additionally, consider implementing a check for the `isDeprecated` flag within the `setIsLiquidateBorrowPaused` function to prevent the pausing of liquidation activities in already deprecated markets. This dual-check mechanism will safeguard against scenarios where market deprecation and liquidation pauses could interact in ways that negatively affect the platform's functionality. Implementing these recommendations will enhance the platform's risk management and ensure a more robust and secure market lifecycle management process.

DLF - Desynchronization Liquidation Flaw

Criticality	Minor / Informative
Location	aave-v2/ExitPositionsManager.sol#L233
Status	Unresolved

Description

The contract is designed to aggregate all individual user positions on the Aave lending platform into a single collective position managed by Morpho. This architecture simplifies interactions with the Aave protocol by pooling user transactions. However, when the aggregated Morpho position on Aave is liquidated due to market conditions, the internal tracking of user positions within Morpho fails to update accordingly. This desynchronization results in users being unable to accurately manage their collateral and debt through Morpho, as the platform's records do not reflect the true state of their investments post-liquidation. Furthermore, this flaw enables a scenario where a position can be liquidated multiple times for profit. In the given example, a user's position, represented on Aave through Morpho, is liquidated, but Morpho's internal records remain unchanged. This discrepancy allows for the possibility of a second liquidation under Morpho's identical liquidation parameters, leading to an exploitative extraction of liquidation bonuses without the user's debt or collateral balances being accurately updated.

```
    function liquidateLogic(
        address _poolTokenBorrowed,
        address _poolTokenCollateral,
        address _borrower,
        uint256 _amount
    ) external {
        ...

        vars.amountToLiquidate = Math.min(
            _amount,
            _getUserBorrowBalanceInOf(_poolTokenBorrowed,
_borrower).percentMul(vars.closeFactor) // Max liquidatable debt.
        );
        ...
        }
```

Recommendation

It is recommended to enhance the synchronization between Morpho and Aave by implementing a mechanism that automatically updates Morpho's internal user position states upon any changes in the Aave position, especially after liquidations. This could involve setting up event listeners for liquidation events on Aave that trigger immediate state updates in Morpho. Additionally, Morpho should introduce pre-liquidation checks to confirm the accuracy of its internal records against the actual state on Aave, preventing redundant or malicious liquidations. This approach will ensure the integrity of user positions and maintain the reliability of Morpho's platform functionalities.

MOPV - Missing Oracle Price Validation

Criticality	Minor / Informative
Location	aave-v2/ExitPositionsManager.sol#L253 aave-v2/MorphoUtils.sol#L281
Status	Unresolved

Description

The contract is designed to facilitate liquidation processes within the Morpho protocol. A component of this functionality is the `liquidateLogic` function, which relies on price data fetched from an oracle to determine the value of the borrowed token and the collateral. This function retrieves `borrowedTokenPrice` and `collateralPrice` from the oracle to calculate the liquidation amounts and eligibility. However, the current implementation does not include checks to validate the prices fetched. Specifically, it does not verify that the returned prices are greater than zero. This lack of validation could lead to scenarios where incorrect or manipulated price data—such as a zero price—could adversely affect the liquidation process, potentially leading to incorrect liquidation outcomes or the exploitation of the protocol.

```
vars.borrowedTokenPrice = oracle.getAssetPrice(tokenBorrowed);
vars.collateralPrice = oracle.getAssetPrice(tokenCollateral);
```

```
vars.underlyingToken = market[vars.poolToken].underlyingToken;
vars.underlyingPrice = oracle.getAssetPrice(vars.underlyingToken);
```

Recommendation

It is recommended to incorporate validation checks for the `borrowedTokenPrice` and `collateralPrice` immediately after they are fetched from the oracle within the `liquidateLogic` function. The contract should revert the transaction if either of these prices is zero, indicating that the oracle has returned invalid price data. Implementing these checks will enhance the security and robustness of the liquidation process by ensuring that all operations are based on valid and reasonable price information. This validation is crucial

for maintaining the integrity of the protocol and protecting against potential oracle manipulation or errors.

MZRC - Missing Zero Repayment Check

Criticality	Minor / Informative
Location	compound/PositionsManager.sol#L461
Status	Unresolved

Description

The contract is designed to facilitate the repayment of debts within theecosystem, by using the `repayLogic` function to manage the repayment process. This function includes several checks to ensure the validity and appropriateness of a repayment transaction. However, `toRepay` is determined as the minimum value between the user's borrow balance in the pool (`_getUserBorrowBalanceInOf(_poolToken, _onBehalf)`) and the specified repayment amount (`_amount`). The function correctly reverts if `_amount` is zero but fails to revert if `toRepay` is calculated to be zero, which occurs when the `_onBehalf` user has no outstanding debt in the pool. This oversight can lead to unnecessary execution of the function, wasting gas and resources when there is, in fact, no debt to repay.

```
    function repayLogic(
        address _poolToken,
        address _repayer,
        address _onBehalf,
        uint256 _amount,
        uint256 _maxGasForMatching
    ) external {
        if (_amount == 0) revert AmountIsZero();
        if (!marketStatus[_poolToken].isCreated) revert
MarketNotCreated();
        if (marketPauseStatus[_poolToken].isRepayPaused) revert
RepayIsPaused();
        if (!userMembership[_poolToken][_onBehalf]) revert
UserNotMemberOfMarket();

        _updateP2PIndexes(_poolToken);
        uint256 toRepay =
Math.min(_getUserBorrowBalanceInOf(_poolToken, _onBehalf), _amount);

        _unsafeRepayLogic(_poolToken, _repayer, _onBehalf, toRepay,
_maxGasForMatching);
    }
```

Recommendation

It is recommended to enhance the `repayLogic` function by adding a check that reverts the transaction if `toRepay` equals zero. This additional validation step will ensure that the function only proceeds when there is a meaningful amount of debt to be repaid, thereby preventing the execution of redundant transactions and the wastage of network resources. Implementing this check will improve the efficiency and robustness of the contract's repayment logic, aligning it more closely with the principle of fail-fast for invalid operations.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	aave-v2/EntryPositionsManager.sol#L101,265 aave-v2/ExitPositionsManager.sol#L493 compound/MorphoGovernance.sol#L249
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
underlyingToken.safeTransferFrom(_from, address(this),
_amount);
...
underlyingToken.safeTransfer(msg.sender, _amount);
...
underlyingToken.safeTransferFrom(_repayer, address(this),
_amount);
```

```
underlyingToken.safeTransferFrom(_from, address(this),
_amount);
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	aave-v2/MorphoGovernance.sol#L168-329 aave-v2/MorphoGovernance.sol#L172-332
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function setMaxSortedUsers(uint256 _newMaxSortedUsers) external
onlyOwner {
        if (_newMaxSortedUsers == 0) revert
MaxSortedUsersCannotBeZero();
        maxSortedUsers = _newMaxSortedUsers;
        emit MaxSortedUsersSet(_newMaxSortedUsers);
    }

    function setDefaultMaxGasForMatching(Types.MaxGasForMatching memory
_defaultMaxGasForMatching)
        external
        onlyOwner
    {
        defaultMaxGasForMatching = _defaultMaxGasForMatching;
        emit DefaultMaxGasForMatchingSet(_defaultMaxGasForMatching);
    }

    function setEntryPositionsManager(IEntryPositionsManager
_entryPositionsManager)
        external
        onlyOwner
    {
        if (address(_entryPositionsManager) == address(0)) revert
ZeroAddress();
        entryPositionsManager = _entryPositionsManager;
        emit EntryPositionsManagerSet(address(_entryPositionsManager));
    }


    function setExitPositionsManager(IExitPositionsManager
_exitPositionsManager)
        external
        onlyOwner
    {
        if (address(_exitPositionsManager) == address(0)) revert
ZeroAddress();
        exitPositionsManager = _exitPositionsManager;
        emit ExitPositionsManagerSet(address(_exitPositionsManager));
    }

    function setInterestRatesManager(IInterestRatesManager
_interestRatesManager)
        external
        onlyOwner
    {
        if (address(_interestRatesManager) == address(0)) revert
ZeroAddress();
        interestRatesManager = _interestRatesManager;
        emit InterestRatesSet(address(_interestRatesManager));
    }
```

```
    function setTreasuryVault(address _treasuryVault) external
onlyOwner {
        treasuryVault = _treasuryVault;
        emit TreasuryVaultSet(_treasuryVault);
    }

    function setReserveFactor(address _poolToken, uint16
_newReserveFactor)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        if (_newReserveFactor > MAX_BASIS_POINTS) revert
ExceedsMaxBasisPoints();
        _updateIndexes(_poolToken);

        market[_poolToken].reserveFactor = _newReserveFactor;
        emit ReserveFactorSet(_poolToken, _newReserveFactor);
    }

    function setP2PIndexCursor(address _poolToken, uint16
_p2pIndexCursor)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        if (_p2pIndexCursor > MAX_BASIS_POINTS) revert
ExceedsMaxBasisPoints();
        _updateIndexes(_poolToken);

        market[_poolToken].p2pIndexCursor = _p2pIndexCursor;
        emit P2PIndexCursorSet(_poolToken, _p2pIndexCursor);
    }

    function setIsSupplyPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isSupplyPaused = _isPaused;
        emit IsSupplyPausedSet(_poolToken, _isPaused);
    }

    function setIsBorrowPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        if (!_isPaused && marketPauseStatus[_poolToken].isDeprecated)
```

```
revert MarketIsDeprecated();
        marketPauseStatus[_poolToken].isBorrowPaused = _isPaused;
        emit IsBorrowPausedSet(_poolToken, _isPaused);
    }

    function setIsWithdrawPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isWithdrawPaused = _isPaused;
        emit IsWithdrawPausedSet(_poolToken, _isPaused);
    }

    function setIsRepayPaused(address _poolToken, bool _isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isRepayPaused = _isPaused;
        emit IsRepayPausedSet(_poolToken, _isPaused);
    }

    function setIsLiquidateCollateralPaused(address _poolToken, bool
_isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isLiquidateCollateralPaused =
_isPaused;
        emit IsLiquidateCollateralPausedSet(_poolToken, _isPaused);
    }

    function setIsLiquidateBorrowPaused(address _poolToken, bool
_isPaused)
        external
        onlyOwner
        isMarketCreated(_poolToken)
    {
        marketPauseStatus[_poolToken].isLiquidateBorrowPaused =
_isPaused;
        emit IsLiquidateBorrowPausedSet(_poolToken, _isPaused);
    }

    function setIsPausedForAllMarkets(bool _isPaused) external
onlyOwner {
        uint256 numberOfMarketsCreated = marketsCreated.length;

        ...
```

```
            }
        }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

RTO - Rewards Transition Oversight

Criticality	Minor / Informative
Location	MatchingEngine.sol#L347
Status	Unresolved

Description

The contract is structured to facilitate the management of user rewards within the pool. A functionality of this contract involves the ability to update or change the rewards manager responsible for distributing rewards to users based on their participation in the pool. However, when a new rewards manager is appointed, the contract does not automatically account for or migrate the existing users' states who are already participating in the pool. This oversight results in these users not being tracked by the new rewards manager, consequently making them ineligible to earn rewards post-transition. This flaw can significantly impact user trust and the platform's overall incentive structure, especially for users with substantial contributions to the pool, as they stand to lose out on rewards they would otherwise be entitled to under the new management.

```
   function _updateSupplierInDS(address _poolToken, address _user)
internal {
        ...

        if (address(rewardsManager) != address(0))
            rewardsManager.accrueUserSupplyUnclaimedRewards(_user,
_poolToken, formerValueOnPool);
    }
```

Recommendation

It is recommended to implement a mechanism for initializing the new rewards manager with the current state of all users already participating in the pool. This initialization should include accurately migrating the users' contribution data to ensure they continue to earn rewards seamlessly after the transition. For a practical and immediate workaround, users with significant pool supplies could be advised to resupply or reborrow a minimal amount to the pool. This action would trigger their registration with the new rewards manager, thereby

re-enabling their rewards accrual. However, for a long-term solution, the contract should be enhanced to automatically perform this migration during the rewards manager transition process, ensuring no user is inadvertently excluded from earning rewards due to administrative changes. This approach will maintain the integrity of the rewards system and uphold user confidence in the platform's management practices.

USF - Unbounded Sorting Functionality

Criticality	Minor / Informative
Location	aave-v2/MorphoGovernance.sol#L168 compoundMorphoGovernance.sol#L172
Status	Unresolved

Description

The `MorphoGovernance` contract is responsible for managing governance parameters within the Morpho protocol, specifically through the MorphoGovernance.sol contract. The `setMaxSortedUsers` function, sets the maximum number of users to be sorted in a data structure for efficient management and querying of user positions. However, there is no upper limit set for the `maxSortedUsers` parameter, potentially leading to gas consumption issues during the sorting process. Operations requiring sorting a large number of users could become prohibitively expensive or even fail due to block gas limit constraints, impacting the protocol's efficiency and scalability. Additionally, while the Aave-v2 implementation includes a check to prevent setting `maxSortedUsers` to zero (MaxSortedUsersCannotBeZero()), this crucial validation is absent in the Compound version of the function. This inconsistency can lead to a scenario where `maxSortedUsers` is set to zero in the Compound version, potentially causing functions that rely on sorting to fail or behave unpredictably.

```
    function setMaxSortedUsers(uint256 _newMaxSortedUsers) external
onlyOwner {
        if (_newMaxSortedUsers == 0) revert
MaxSortedUsersCannotBeZero();
        maxSortedUsers = _newMaxSortedUsers;
        emit MaxSortedUsersSet(_newMaxSortedUsers);
    }
```

```
    function setMaxSortedUsers(uint256 _newMaxSortedUsers) external
onlyOwner {
        maxSortedUsers = _newMaxSortedUsers;
        emit MaxSortedUsersSet(_newMaxSortedUsers);
    }
```

Recommendation

It is recommended to introduce an upper bounds limit for the `maxSortedUsers` parameter to prevent potential gas issues associated with sorting a large number of users. This limit should be carefully chosen to balance the need for efficient data management with the constraints of gas costs and blockchain performance. Additionally, to ensure consistency and reliability across different versions of the protocol, the `MaxSortedUsersCannotBeZero()` check should be implemented in the Compound version of the `setMaxSortedUsers` function, mirroring the safeguard present in the Aave-v2 version. Implementing these recommendations will enhance the robustness, efficiency, and consistency of the Morpho protocol's governance mechanisms, ensuring smooth operation and scalability.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	aave-v2/MorphoStorage.sol#L38,39,64,65,66,68,69,70,71,72,73 compound/MorphoStorage.sol#L26,27,54,55,56,57,58,59,60,61,66
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
bool public isClaimRewardsPaused
uint256 public maxSortedUsers
ILendingPoolAddressesProvider public addressesProvider
address public aaveIncentivesController
ILendingPool public pool
IEntryPositionsManager public entryPositionsManager
IExitPositionsManager public exitPositionsManager
IInterestRatesManager public interestRatesManager
address public incentivesVault
address public rewardsManager
address public treasuryVault
```

```
uint256 public maxSortedUsers
uint256 public dustThreshold
IPositionsManager public positionsManager
address public incentivesVault
IRewardsManager public rewardsManager
IInterestRatesManager public interestRatesManager
IComptroller public comptroller
address public treasuryVault
address public cEth
address public wEth
bool public isClaimRewardsPaused
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	aave-v2/MorphoUtils.sol#L66,88,89,90,104 aave-v2/MorphoGovernance.sol#L142,143,144,145,146,147,168,176,186,197,208,219,227,242,257,269,282,294,306,318,329,346,358,373,386,417,418,419 aave-v2/Morpho.sol#L21,31,32,33,46,47,48,49,57,66,67,68,76,85,86,87,96,106,107,108,119,120,121,122 aave-v2/InterestRatesManager.sol#L53 aave-v2/ExitPositionsManager.sol#L148,149,150,151,152,175,176,177,178,179,198,199,200,201,288 aave-v2/EntryPositionsManager.sol#L84,85,86,87,88,183,184,185 compound/RewardsManager.sol#L57,67,78,91,105,106,107,118,119,120 compound/PositionsManager.sol#L235,236,237,238,239,334,335,336,435,436,437,438,439,462,463,464,465,466,485,486,487,488,549 compound/MorphoUtils.sol#L42,56,78,79,80,95 compound/MorphoGovernance.sol#L144,145,146,147,148,149,150,151,172,179,189,197,208,215,222,230,245,260,272,285,297,309,321,332,349,360,368,383,396,425 compound/Morpho.sol#L33,43,44,45,58,59,60,61,69,78,79,80,88,97,98,99,108,118,119,120,131,132,133,134,151 compound/interfaces/IMorpho.sol#L15,16,17 compound/InterestRatesManager.sol#L52
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.

5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
Types.PositionType _positionType
address _poolToken
address _user
IEntryPositionsManager _entryPositionsManager
IExitPositionsManager _exitPositionsManager
IInterestRatesManager _interestRatesManager
ILendingPoolAddressesProvider _lendingPoolAddressesProvider
Types.MaxGasForMatching memory _defaultMaxGasForMatching
uint256 _maxSortedUsers
uint256 _newMaxSortedUsers
address _treasuryVault
uint16 _newReserveFactor
uint16 _p2pIndexCursor
bool _isPaused

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	aave-v2/MorphoUtils.sol#L270,275 aave-v2/MorphoGovernance.sol#L332,394 aave-v2/MatchingEngine.sol#L70,73,131,134,191,194,252,255 aave-v2/ExitPositionsManager.sol#L224,338,494 aave-v2/EntryPositionsManager.sol#L104,202 compound/RewardsManager.sol#L138 compound/PositionsManager.sol#L252,349,505,598,761,1012 compound/MorphoUtils.sol#L130 compound/MorphoGovernance.sol#L335,404 compound/MatchingEngine.sol#L74,77,90,136,139,154,201,204,217,263,266,281
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
LiquidityVars memory vars
uint256 i
MatchVars memory vars
address firstPoolSupplier
UnmatchVars memory vars
address firstP2PSupplier
address firstPoolBorrower
address firstP2PBorrower
LiquidateVars memory vars
WithdrawVars memory vars
RepayVars memory vars
SupplyVars memory vars
uint256 toWithdraw
```

```
uint256 i
SupplyVars memory vars
uint256 toWithdraw
LiquidateVars memory vars
WithdrawVars memory vars
RepayVars memory vars
uint256 index
MatchVars memory vars
address firstPoolSupplier
uint256 poolSupplyBalance
UnmatchVars memory vars
address firstP2PSupplier
uint256 p2pSupplyBalance
address firstPoolBorrower
...
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	aave-v2/MorphoGovernance.sol#L220 compound/MorphoGovernance.sol#L164,165,216
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
treasuryVault = _treasuryVault
```

```
cEth = _cEth
wEth = _wEth
treasuryVault = _treasuryVault
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	RewardsDistributor.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L22 - Potential Locked Ether

Criticality	Minor / Informative
Location	compound/Morpho.sol#L168
Status	Unresolved

Description

The contract contains Ether that has been placed into a Solidity contract and is unable to be transferred. Thus, it is impossible to access the locked Ether. This may produce a financial loss for the users that have called the payable method.

```
receive() external payable {}
```

Recommendation

The team is advised to either remove the payable method or add a withdraw functionality. it is important to carefully consider the risks and potential issues associated with locked Ether.

Functions Analysis

Aave-v2

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
RewardsManager	Implementation	IRewardsManager, Initializable		
		Public	✓	initializer
	initialize	External	✓	initializer
	getLocalCompSupplyState	External		-
	getLocalCompBorrowState	External		-
	claimRewards	External	✓	onlyMorpho
	accrueUserSupplyUnclaimedRewards	External	✓	onlyMorpho
	accrueUserBorrowUnclaimedRewards	External	✓	onlyMorpho
	_accrueUserUnclaimedRewards	Internal	✓	
	_accrueSupplierComp	Internal	✓	
	_accrueBorrowerComp	Internal	✓	
	_updateSupplyIndex	Internal	✓	
	_updateBorrowIndex	Internal	✓	
PositionsManager	Implementation	IPositionsManager, MatchingEngine		
	supplyLogic	External	✓	-

	borrowLogic	External	✓	-
	withdrawLogic	External	✓	-
	repayLogic	External	✓	-
	liquidateLogic	External	✓	-
	increaseP2PDeltasLogic	External	✓	isMarketCreated
	_unsafeWithdrawLogic	Internal	✓	
	_unsafeRepayLogic	Internal	✓	
	_supplyToPool	Internal	✓	
	_withdrawFromPool	Internal	✓	
	_borrowFromPool	Internal	✓	
	_repayToPool	Internal	✓	
	_enterMarketIfNeeded	Internal	✓	
	_leaveMarketIfNeeded	Internal	✓	
	_liquidationAllowed	Internal		
MorphoUtils	Implementation	MorphoStorage		
	getEnteredMarkets	External		-
	getAllMarkets	External		-
	getHead	External		-
	getNext	External		-
	updateP2PIndexes	External	✓	isMarketCreated
	_updateP2PIndexes	Internal	✓	
	_isLiquidatable	Internal		

	_getUserLiquidityDataForAsset	Internal		
	_getUserSupplyBalanceInOf	Internal		
	_getUserBorrowBalanceInOf	Internal		
	_getUnderlying	Internal		
MorphoStorage	Implementation	OwnableUpgradeable, ReentrancyGuardUpgradeable		
		Public	✓	initializer
MorphoGovernance	Implementation	MorphoUtils		
	initialize	External	✓	initializer
	setMaxSortedUsers	External	✓	onlyOwner
	setDefaultMaxGasForMatching	External	✓	onlyOwner
	setPositionsManager	External	✓	onlyOwner
	setInterestRatesManager	External	✓	onlyOwner
	setRewardsManager	External	✓	onlyOwner
	setTreasuryVault	External	✓	onlyOwner
	setDustThreshold	External	✓	onlyOwner
	setReserveFactor	External	✓	onlyOwner isMarketCreated
	setP2PIndexCursor	External	✓	onlyOwner isMarketCreated
	setIsSupplyPaused	External	✓	onlyOwner isMarketCreated

	setIsBorrowPaused	External	✓	onlyOwner isMarketCreated
	setIsWithdrawPaused	External	✓	onlyOwner isMarketCreated
	setIsRepayPaused	External	✓	onlyOwner isMarketCreated
	setIsLiquidateCollateralPaused	External	✓	onlyOwner isMarketCreated
	setIsLiquidateBorrowPaused	External	✓	onlyOwner isMarketCreated
	setIsPausedForAllMarkets	External	✓	onlyOwner
	setIsP2PDisabled	External	✓	onlyOwner isMarketCreated
	setIsClaimRewardsPaused	External	✓	onlyOwner
	setIsDeprecated	External	✓	onlyOwner isMarketCreated
	increaseP2PDeltas	External	✓	onlyOwner
	claimToTreasury	External	✓	onlyOwner
	createMarket	External	✓	onlyOwner
	_setPauseStatus	Internal	✓	
Morpho	Implementation	MorphoGovernance		
	supply	External	✓	nonReentrant
	supply	External	✓	nonReentrant
	supply	External	✓	nonReentrant
	borrow	External	✓	nonReentrant

	borrow	External	✓	nonReentrant
	withdraw	External	✓	nonReentrant
	withdraw	External	✓	nonReentrant
	repay	External	✓	nonReentrant
	repay	External	✓	nonReentrant
	liquidate	External	✓	nonReentrant
	claimRewards	External	✓	nonReentrant
		External	Payable	-
	_supply	Internal	✓	
	_borrow	Internal	✓	
	_withdraw	Internal	✓	
	_repay	Internal	✓	
MatchingEngine	Implementation	MorphoUtils		
	_matchSuppliers	Internal	✓	
	_unmatchSuppliers	Internal	✓	
	_matchBorrowers	Internal	✓	
	_unmatchBorrowers	Internal	✓	
	_updateSupplierInDS	Internal	✓	
	_updateBorrowerInDS	Internal	✓	
InterestRatesManager	Implementation	IInterestRatesManager, MorphoStorage		

	updateP2PIndexes	External	✓	-
	_computeP2PIndexes	Internal		

Common

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
RewardsDistributor	Implementation	Ownable		
		Public	✓	-
	updateRoot	External	✓	onlyOwner
	withdrawMorphoTokens	External	✓	onlyOwner
	claim	External	✓	-

Compound

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
RewardsManager	Implementation	IRewardsManager, Initializable		
		Public	✓	initializer
	initialize	External	✓	initializer
	getLocalCompSupplyState	External		-
	getLocalCompBorrowState	External		-
	claimRewards	External	✓	onlyMorpho
	accrueUserSupplyUnclaimedRewards	External	✓	onlyMorpho
	accrueUserBorrowUnclaimedRewards	External	✓	onlyMorpho
	_accrueUserUnclaimedRewards	Internal	✓	
	_accrueSupplierComp	Internal	✓	
	_accrueBorrowerComp	Internal	✓	
	_updateSupplyIndex	Internal	✓	
	_updateBorrowIndex	Internal	✓	
PositionsManager	Implementation	IPositionsManager, MatchingEngine		
	supplyLogic	External	✓	-
	borrowLogic	External	✓	-

	withdrawLogic	External	✓	-
	repayLogic	External	✓	-
	liquidateLogic	External	✓	-
	increaseP2PDeltasLogic	External	✓	isMarketCreated
	_unsafeWithdrawLogic	Internal	✓	
	_unsafeRepayLogic	Internal	✓	
	_supplyToPool	Internal	✓	
	_withdrawFromPool	Internal	✓	
	_borrowFromPool	Internal	✓	
	_repayToPool	Internal	✓	
	_enterMarketIfNeeded	Internal	✓	
	_leaveMarketIfNeeded	Internal	✓	
	_liquidationAllowed	Internal		
MorphoUtils	Implementation	MorphoStorage		
	getEnteredMarkets	External		-
	getAllMarkets	External		-
	getHead	External		-
	getNext	External		-
	updateP2PIndexes	External	✓	isMarketCreated
	_updateP2PIndexes	Internal	✓	
	_isLiquidatable	Internal		
	_getUserLiquidityDataForAsset	Internal		

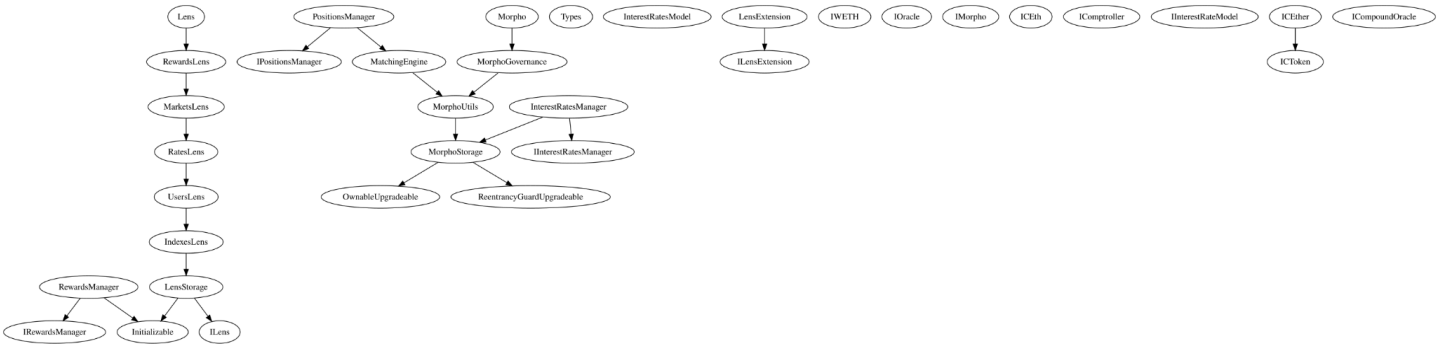
	_getUserSupplyBalanceInOf	Internal		
	_getUserBorrowBalanceInOf	Internal		
	_getUnderlying	Internal		
MorphoStorage	Implementation	OwnableUpgradeable, ReentrancyGuardUpgradeable		
		Public	✓	initializer
MorphoGovernance	Implementation	MorphoUtils		
	initialize	External	✓	initializer
	setMaxSortedUsers	External	✓	onlyOwner
	setDefaultMaxGasForMatching	External	✓	onlyOwner
	setPositionsManager	External	✓	onlyOwner
	setInterestRatesManager	External	✓	onlyOwner
	setRewardsManager	External	✓	onlyOwner
	setTreasuryVault	External	✓	onlyOwner
	setDustThreshold	External	✓	onlyOwner
	setReserveFactor	External	✓	onlyOwner isMarketCreated
	setP2PIndexCursor	External	✓	onlyOwner isMarketCreated
	setIsSupplyPaused	External	✓	onlyOwner isMarketCreated

	setIsBorrowPaused	External	✓	onlyOwner isMarketCreated
	setIsWithdrawPaused	External	✓	onlyOwner isMarketCreated
	setIsRepayPaused	External	✓	onlyOwner isMarketCreated
	setIsLiquidateCollateralPaused	External	✓	onlyOwner isMarketCreated
	setIsLiquidateBorrowPaused	External	✓	onlyOwner isMarketCreated
	setIsPausedForAllMarkets	External	✓	onlyOwner
	setIsP2PDisabled	External	✓	onlyOwner isMarketCreated
	setIsClaimRewardsPaused	External	✓	onlyOwner
	setIsDeprecated	External	✓	onlyOwner isMarketCreated
	increaseP2PDeltas	External	✓	onlyOwner
	claimToTreasury	External	✓	onlyOwner
	createMarket	External	✓	onlyOwner
	_setPauseStatus	Internal	✓	
Morpho	Implementation	MorphoGovernance		
	supply	External	✓	nonReentrant
	supply	External	✓	nonReentrant
	supply	External	✓	nonReentrant
	borrow	External	✓	nonReentrant

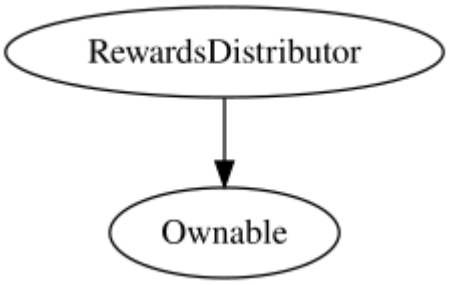
	borrow	External	✓	nonReentrant
	withdraw	External	✓	nonReentrant
	withdraw	External	✓	nonReentrant
	repay	External	✓	nonReentrant
	repay	External	✓	nonReentrant
	liquidate	External	✓	nonReentrant
	claimRewards	External	✓	nonReentrant
		External	Payable	-
	_supply	Internal	✓	
	_borrow	Internal	✓	
	_withdraw	Internal	✓	
	_repay	Internal	✓	
MatchingEngine	Implementation	MorphoUtils		
	_matchSuppliers	Internal	✓	
	_unmatchSuppliers	Internal	✓	
	_matchBorrowers	Internal	✓	
	_unmatchBorrowers	Internal	✓	
	_updateSupplierInDS	Internal	✓	
	_updateBorrowerInDS	Internal	✓	

Inheritance Graph

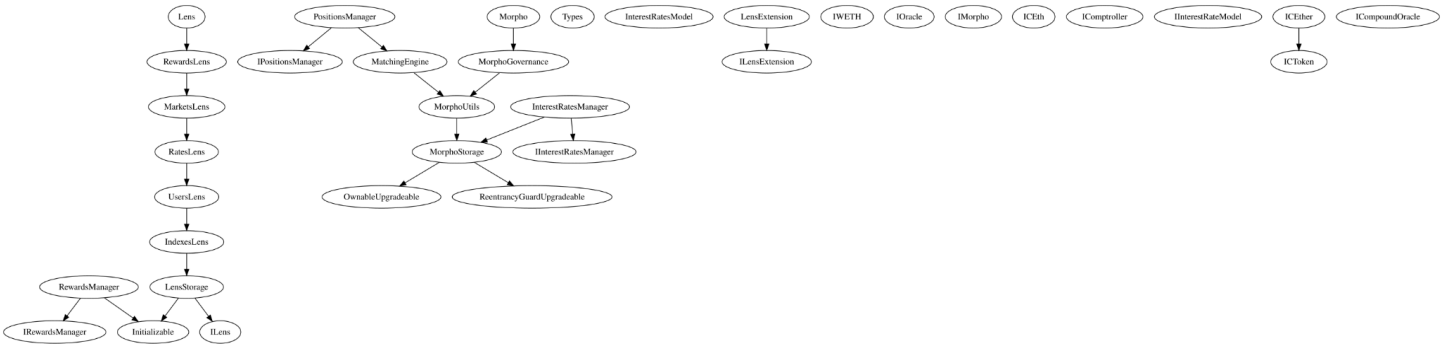
Aave-v2



Common

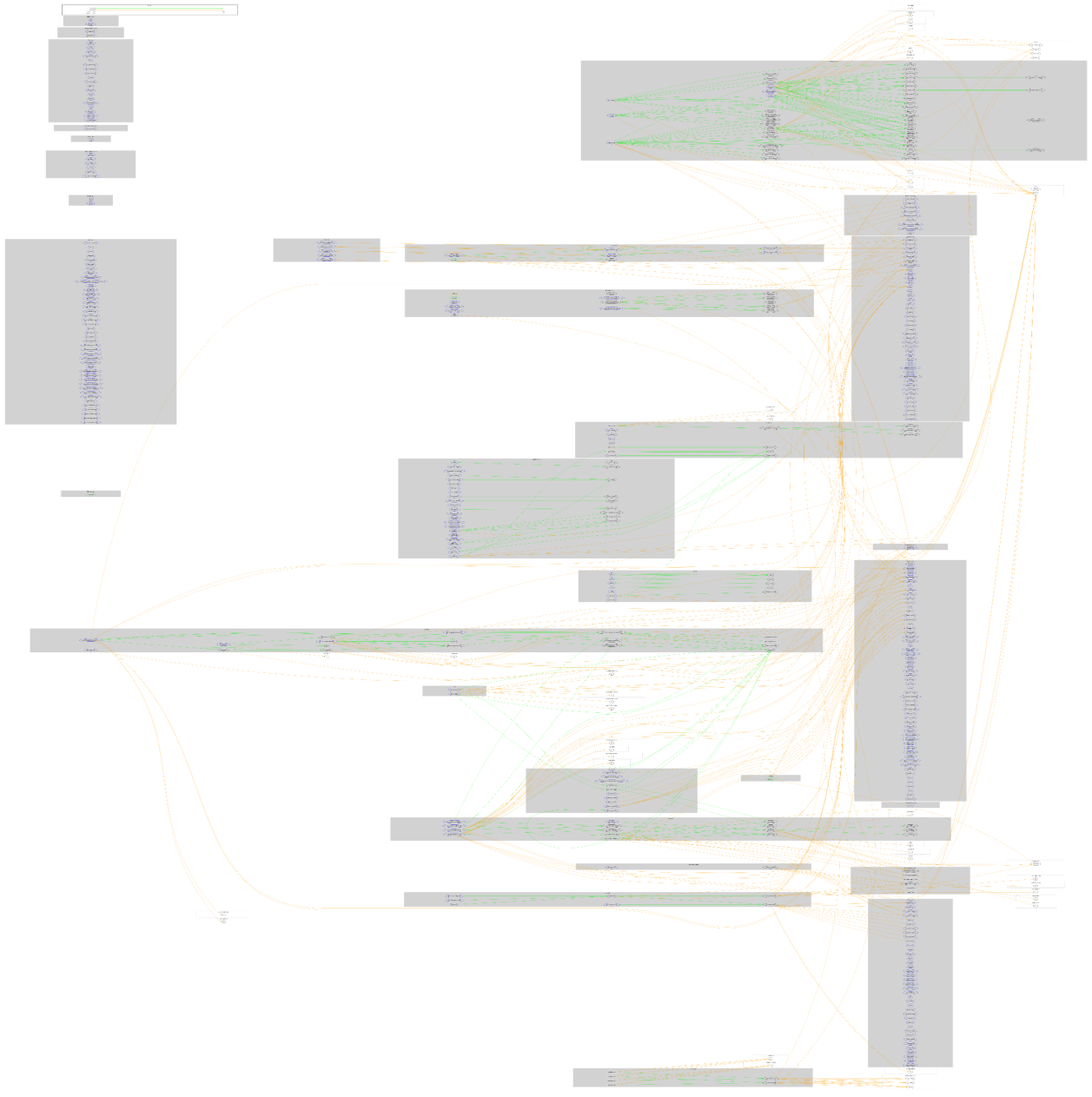


Compound

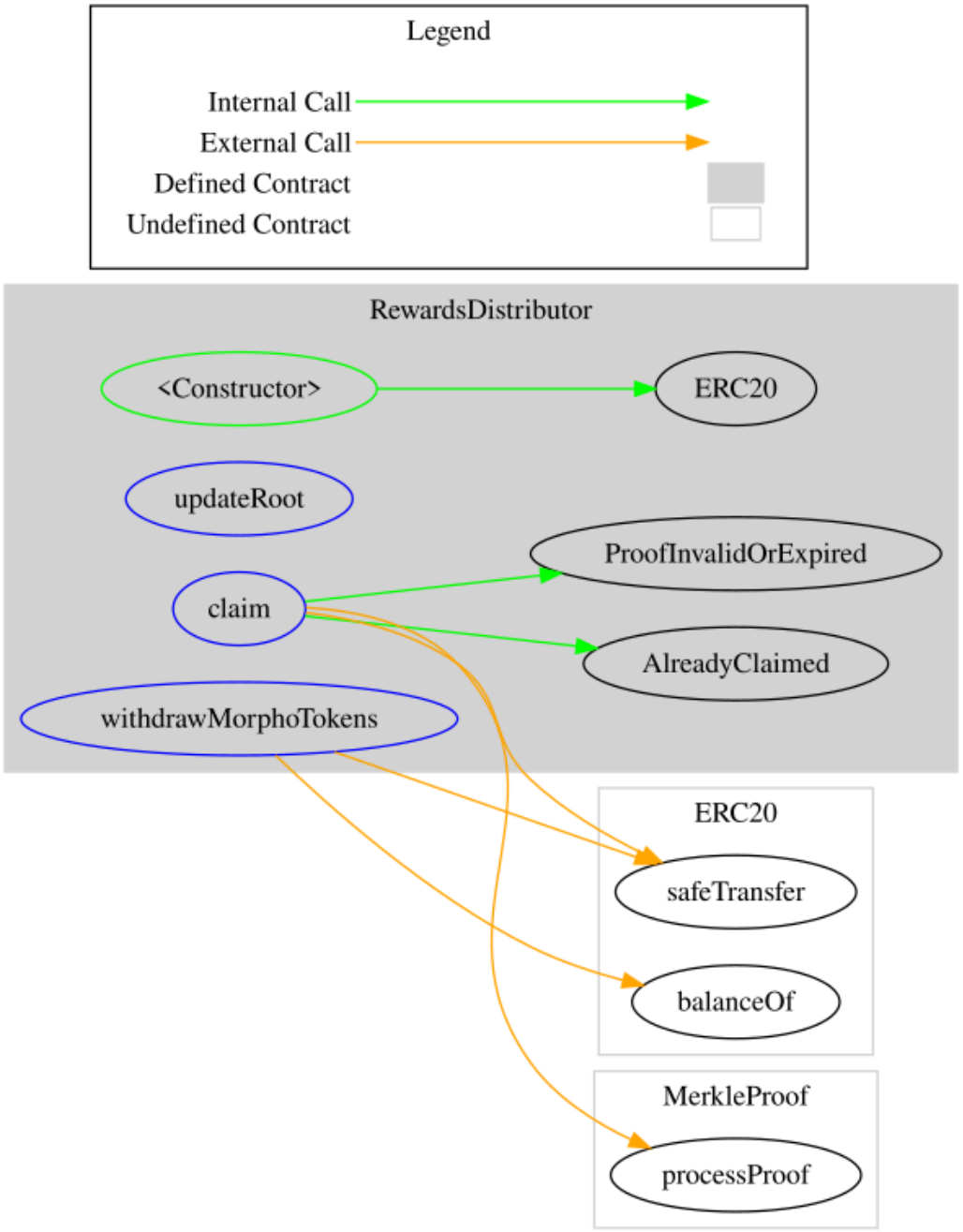

Lens
PositionsManager
Morpho
Types
InterestRatesModel
LensExtension
IWETH
IOracle
IMorpho
ICEth
IComptroller
IInterestRateModel
ICEther
ICompoundOracle
RewardsLens
IPositionsManager
MatchingEngine
MorphoGovernance
ILensExtension
ICToken
MarketsLens
MorphoUtils
InterestRatesManager
RatesLens
MorphoStorage
IInterestRatesManager
UsersLens
OwnableUpgradeable
ReentrancyGuardUpgradeable
IndexesLens
RewardsManager
LensStorage
IRewardsManager
Initializable
ILens

Flow Graph

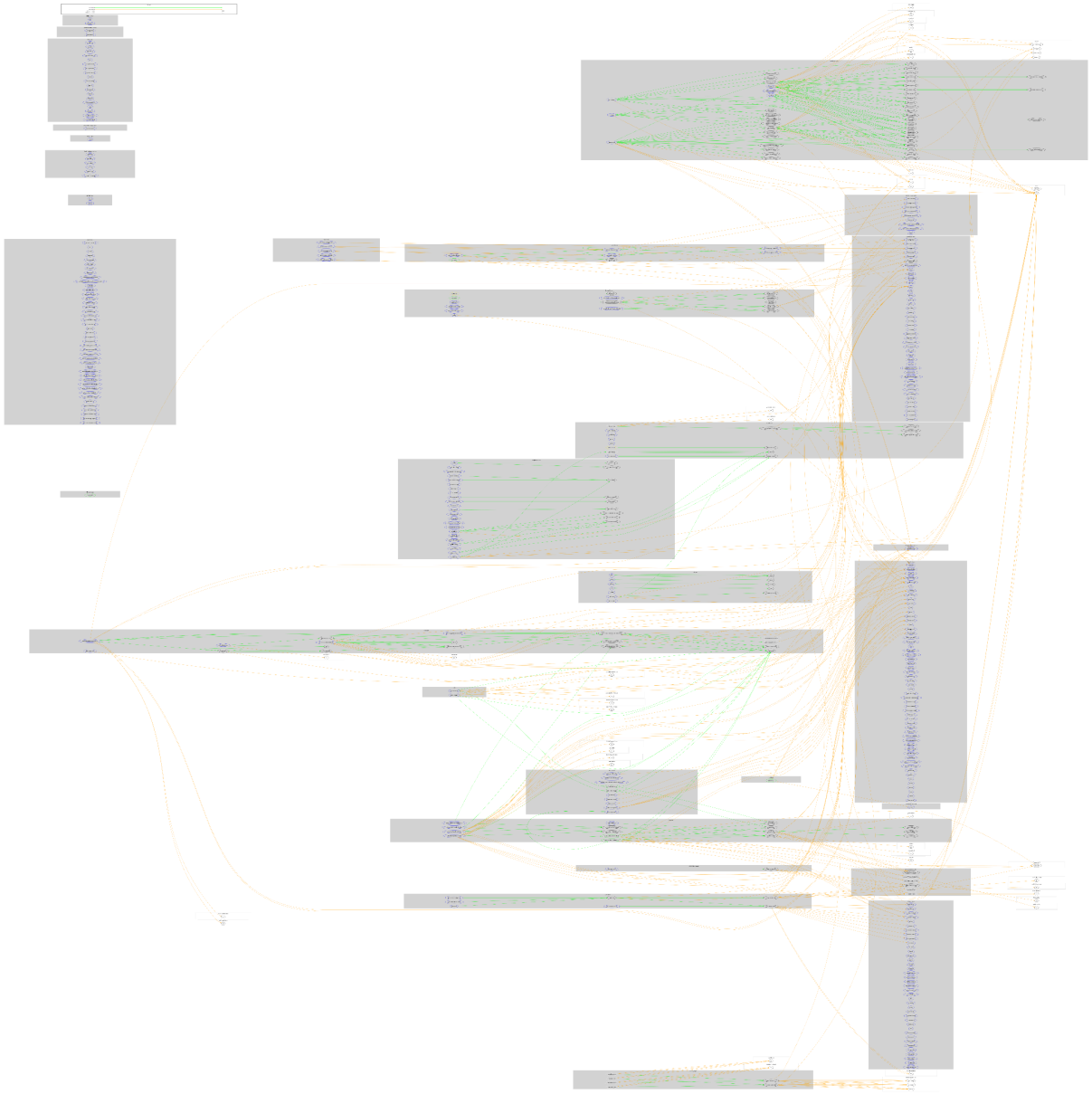
Aave-v2



Common


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
RewardsDistributor
<Constructor>
ERC20
updateRoot
ProofInvalidOrExpired
claim
AlreadyClaimed
withdrawMorphoTokens
ERC20
safeTransfer
balanceOf
MerkleProof
processProof

Compound



Summary

Hakura Protocol contract implements a a comprehensive DeFi application. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io