PUBLIC

Code Assessment

of the Morpho (Aave v3) Smart Contracts

September 27, 2022

Produced for

Morpho

by

CHAINSECURITY

Contents

1 Executive Summary 3

2 Assessment Overview 5

3 Limitations and use of report 9

4 Terminology 10

5 Findings 11

6 Resolved Findings 16

7 Notes 23

1 Executive Summary

Dear Morpho Team,

Thank you for trusting us to help Morpho Labs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Morpho (Aave v3) according to Scope to support you in forming an opinion on their security risks.

Morpho Labs implements a peer-to-peer lending protocol that leverages the liquidity of existing lending protocols like Aave or Compound to allow instant withdrawals. Peer-to-peer matched users benefit from better rates than users of the underlying lending protocols.

The most critical subjects covered in our audit are access control, functional correctness and precision of arithmetic operations. Access control is extensive. Functional correctness of the main contracts is high. Functional correctness of the `HeapOrdering` data structure is not sufficient as the Heap data structure can be spammed. This issue can also lead to accidental violation of the Heap ordering, causing users additional gas fees. Precision of arithmetic operations is high.

The general subjects covered are documentation and gas efficiency. Documentation is extensive. Gas efficiency is improvable as shown in Gas inefficiencies.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Risk Accepted	1
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	21
• Code Corrected	16
• Code Partially Corrected	3
• Risk Accepted	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Morpho (Aave v3) repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

morpho-contracts

V	Date	Commit Hash	Note
1	27 June 2022	3653fbc4db9037974fd42e17eab49404297c69e3	Initial Version
2	12 August 2022	e42998059cfc9a82a851685569385a20c8a73825	Second Version
3	19 September 2022	e3a9755a93c798c1e4a9039f93617d4ea85eab96	Third Version
4	26 September 2022	81afc946e08deced6437f2345f6e6abea547748a	Third Version

morpho-data-structures

V	Date	Commit Hash	Note
1	07 June 2022	83577ec6bdc18fc3c83b4a15969e0582a07c287e	Initial Version

morpho-utils

V	Date	Commit Hash	Note
1	12 August 2022	cb43513a1a2d6b2eed485b7e58a9724255ca417d	Initial Version

For the solidity smart contracts, the compiler version `0.8.10` was chosen (for compatibility with Aave contracts).

2.1.1 *Included in scope*

The scope of the aforementioned repositories is limited to:

2.1.2 *morpho-contracts*

- All files in the `contracts/aave-v3` folder except `Lens.sol`.
- `common/rewards-distribution/RewardsDistributor.sol`.

2.1.3 *morpho-data-structures*

- `contracts/HeapOrdering.sol`

2.1.4 *morpho-utils*

- All files in `src/math` except `CompoundMath.sol`.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Morpho Labs offers a peer-to-peer lending protocol that allows direct matches between suppliers and borrowers of certain tokens. It builds upon existing lending protocols like **Compound** or **AAVE** and utilizes these protocols' liquidity in order to allow users to enter / exit markets even when no other peers are available as counterparty.

Morpho (Aave v3) specifically builds upon the AAVE v3 lending protocol. The amounts users want to borrow or supply are saved into partially ordered **Heap** data structures. If no counter-party for a supplier / borrower is found, the protocol leverages the underlying lending pool to match the request. Users matched to this lending pool will pay the underlying pool's interest rates and are now available for peer-to-peer matching. As soon as another user's request can be matched to one or multiple users on the underlying lending pool, the positions are moved to peer-to-peer and all participating users now benefit from Morpho (Aave v3)'s improved rates which always reside in the spread between the underlying lending pool's supply and borrow rates.

2.2.1 Morpho

`Morpho` is the main contract that all state-changing user actions take place through. Users can `supply` funds to earn interest or deposit collateral, `borrow` funds, `withdraw` their supplied funds and `repay` borrowed funds. Additionally, any user can `liquidate` borrow positions that are under water and users that have been matched to the underlying pool can `claimRewards` of the reward tokens that are distributed to Morpho (Aave v3)'s position on this pool.

Due to Ethereum's contract size restrictions, `Morpho` performs `delegatecall` calls to external contracts that define the logic for the respective functions (with the exception of `claimRewards`):

- `supply` and `borrow` call logic in the `EntryPositionsManager`.
- `withdraw`, `repay` and `liquidate` call logic in the `ExitPositionsManager`.

Furthermore, the contract exposes governance functions that allow for the update of callable contract addresses and parameters. A function `increaseP2PDeltas` can be used to convert matched P2P users back to the underlying pool. Market tokens are approved with `type(uint256).max` to the underlying Aaave contract on market creation.

2.2.2 EntryPositionsManager

The `EntryPositionsManager` contains the logic for supplying and borrowing. Because matching users peer-to-peer requires their positions to be updated in a sorted `Heap` data structure, gas costs for these transactions can increase dramatically depending on the state of the data structures. For this reason, users can call the functions with a gas limit for matching other peer-to-peer users. If the limit is exceeded, the remaining amount that has not yet been matched peer-to-peer is matched with the underlying pool instead.

Both `supplyLogic` and `borrowLogic` functions follow the same scheme:

- Reduce borrow / supply `delta` and repay / withdraw the amounts to / from the underlying pool (this mechanism is explained in the next section).
- Match borrowers / suppliers and repay / withdraw the amounts to / from the underlying pool.
- Supply / borrow the remaining amounts to / from the underlying pool.

2.2.3 ExitPositionsManager

The ExitPositionsManager contains the logic for withdrawing, repaying and liquidating. Contrary to the supply and borrow logic, users cannot choose a gas limit because there is no incentive for choosing anything other than 0. Instead, a constant gas limit is used for all functions. Because it is possible that this gas limit is also exceeded, Morpho (Aave v3) employs a delta mechanism. If it is not possible to unmatch all peer-to-peer connections of a user, there is a mismatch between the amounts that are in peer-to-peer and on pool: Some tokens that are still registered as peer-to-peer connections in Morpho (Aave v3)'s ledger are now on the pool. The delta contains these amounts and makes sure that the pool rate that has to be payed is evenly split amongst all peer-to-peer users.

The functions withdrawLogic and repayLogic follow the same scheme:

- Withdraw / repay all available amounts for the user from / to the underlying pool.
- Reduce supply / borrow delta and withdraw / repay them from / to the pool.
- (Only repayLogic) Remove the fee (spread between peer-to-peer supply and borrow rate) and keep the amount on the contract.
- Match suppliers / borrowers and withdraw / repay the amounts from / to the underlying pool.

liquidateLogic combines the logic by first repaying the debt of the account that is under water and then withdrawing the account's collateral to the liquidator. Liquidation is only possible if a borrower falls below a certain health factor. The health factor is determined by a liquidation threshold that is provided by the underlying pool for each token. Similarly, withdrawals are only possible if the user health factor is still above the threshold after the withdrawal is completed.

2.2.4 InterestRateManager

All Morpho functions update the interest rate indices before performing any actions. Indices are updated once per block and are set in the spread between the supply and borrow rate of the underlying pool depending on the reserveFactor (spread between the Morpho indices) and the p2pIndexCursor (position between the indices of the underlying pool). For this purpose, the InterestRateManager contains a function updateIndexes which is called via delegatecall from other contracts.

2.2.5 RewardsManager

The RewardsManager handles the rewards AAVE distributes to Morpho (Aave v3)'s account. The rewards are distributed to each user that holds any position on the underlying pool since the last update.

RewardsManager.claimRewards is called from Morpho.claimRewards which can be called by any user. The rest of the functions are copies of AAVE v3's RewardsController that have been slightly changed so that Morpho (Aave v3)'s position on AAVE can be used as the data source.

2.2.6 IncentivesVault

Users who wish to trade their claimed rewards for MORPHO tokens (not yet available at the time of this writing) can call Morpho.claimRewards with a parameter that enables sending the rewards to the IncentivesVault where the tokens are traded for an equivalent value of MORPHO tokens with a small bonus.

2.2.7 RewardsDistributor

Apart from the rewards of the underlying protocol, Morpho (Aave v3) also aims to distribute MORPHO tokens as incentive for the use of the protocol. The RewardsDistributor allows users to claim rewards based on their usage of the contracts. The rules will be defined by the DAO (not yet available at the time of this writing) and a Merkle-Tree will be constructed off-chain following these rules. The root of the tree will then be added to the RewardsDistributor contract once a month and users can claim their rewards using the claim function with a Merkle-Proof of their account balance.

2.2.8 Roles & Trust Model

The main `Morpho` contract and the `RewardsManager` contract are deployed as `TransparentUpgradeableProxy` with the admin address set to a `ProxyAdmin` contract. The contract's `initializer` then registers the caller as the owner of the contract. The owner of the `ProxyAdmin` and the proxy itself has comprehensive power over the contracts. They can:

- Upgrade the contracts to any new implementation.
- Change the addresses of the contracts that are called via `delegatecall`.
- Change the parameters of the contracts.
- Transfer the protocol reserves to another address even when the associated markets are paused.

All other contracts are directly deployed and have corresponding setter methods in the proxied contracts.

Morpho Labs claims to establish a DAO contract that will take ownership of Morpho (Aave v3)'s contracts in the future.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	1

- Heap Data Structure Can Be Spammed Risk Accepted

Medium-Severity Findings	1

- Interfaces Not Implemented / Available Risk Accepted

Low-Severity Findings	5

- Ambiguous Naming Code Partially Corrected
- Gas Inefficiencies Code Partially Corrected
- Missing Sanity Checks Code Partially Corrected
- Rewards Can Be Withdrawn by Admins Risk Accepted
- Variable Shadowing Risk Accepted

5.1 Heap Data Structure Can Be Spammed

Correctness High Version 1 Risk Accepted

The users' supply and borrow information is stored in Heap data structures. The parameter `_maxSortedUsers` sets the maximum sorted user amount in order to limit the gas spent on updating the Heap. The data structure would halve the length of the Heap when `maxSortedUsers` is exceeded. However, this behavior would potentially put an incoming user to a higher priority than an existing one. This behavior can be abused by bad actors to fill the ordered portion of the Heap with dust:

Consider the following example:

- `maxSortedUsers` is set to 4.
- **Step 1:** User 1 and user 2 are legitimate users that supplied 400 and 300 tokens respectively.
- **Step 2:** An attacker now supplies 600, 500 and 1 token with three different addresses.
- **Step 3:** The attacker withdraws 599 and 499 tokens from accounts 3 and 4.

The described behavior is detailed in **Figure 1**. Blue boxes show accounts in the ordered portion of the Heap, green boxes show accounts in the non-ordered portion.

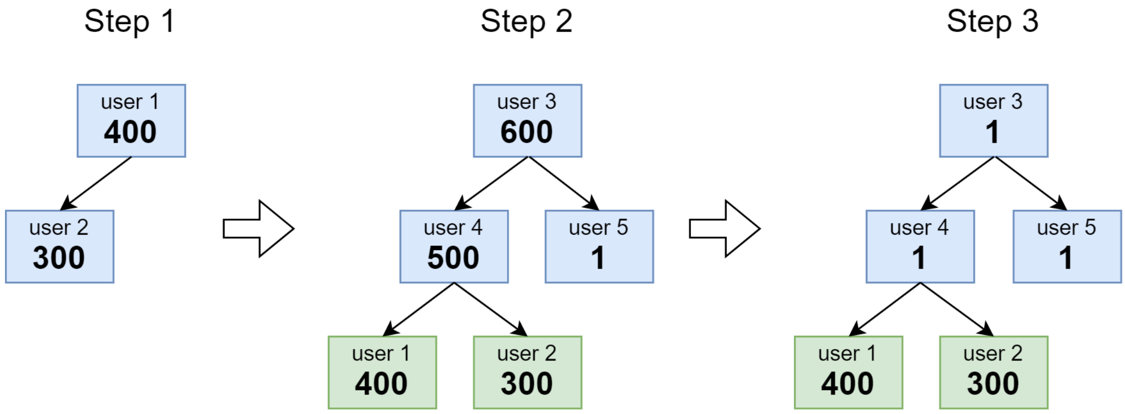

Figure 1: *Spam attack on the Heap*

As a result, the supplied liquidity of users 1 and 2 is now only reachable after the dust of the attacker's accounts has been matched.

Risk accepted:

Morpho Labs accepts the risk with the following statement:

> We know that the heap structure has still some drawbacks (even if it's better than the double linked list implemented on the compound contracts) and we acknowledge the manipulation of the heap. The spam attack is likely to be costly to conduct, moreover, if users come after with greater amounts the dust accounts will be pushed outside the heap.

5.2 Interfaces Not Implemented / Available

Correctness Medium Version 1 Risk Accepted

`Morpho` does not extend the `IMorpho` interface. This can lead to errors during development and integration by third parties as the interface might not match up with the implementations. Indeed, the `IMorpho` interface lacks some public functions like `setInterestRates` or `incentivesVault`.

Risk accepted:

Morpho Labs accepts the risk and tries to maintain correct interfaces.

5.3 Ambiguous Naming

Design Low Version 1 Code Partially Corrected

The function `MorphoGovernance.setInterestRates` is a setter for the `InterestRateManager` address, contrary to the function name which implies it sets interest rate values.

Code partially corrected:

`MorphoGovernance.setInterestRates` has been renamed to `MorphoGovernance.setInterestRateManager` but still emits an event `InterestRatesSet` with ambiguous naming.

5.4 Gas Inefficiencies

Design Low Version 1 Code Partially Corrected

Gas efficiency can be improved in several places:

- The underlying token of AAVE's aTokens is fetched in some functions (e.g. `EntryPositionsManager.supplyLogic`). The addresses could be cached in the contract's storage instead to avoid unnecessary external calls.
- Redundant storage reads are performed in some functions. The values could be cached in the stack or in memory:
    - `delta.p2pBorrowDelta` is read multiple times in `EntryPositionsManager.supplyLogic`.
    - `userMarkets[_user]` is read in every iteration of functions that call `MorphoUtils._isSupplyingOrBorrowing`.
    - Many more examples can be found.
- Redundant storage writes are performed in some functions. The values could be updated in a stack or memory variable and written to storage at the end.
    - `ExitPositionsManager._safeRepayLogic` updates `delta.p2pBorrowAmount` up to 3 times.
    - `ExitPositionsManager._safeRepayLogic` updates `delta.p2pSupplyAmount` up to 2 times.
- Redundant external calls are performed in some functions. The values could be cached in the stack or memory and passed to other called functions:
    - `IAToken.UNDERLYING_ASSET_ADDRESS` is called in `EntryPositionsManager.borrowLogic` and in the sub-call to `_borrowAllowed`.
    - Calls to `pool.getConfiguration` in `ExitPositionsManager.liquidateLogic` are already performed by the sub-call to `_liquidationAllowed`.
    - `MorphoGovernance.createMarket` calls `pool.getConfiguration` and then `pool.getReserveData` which also contains the configuration.
    - In `MorphoGovernance.createMarket`, the values retrieved from `pool.getReserveNormalizedIncome` and `pool.getReserveNormalizedVariableDebt` could be computed from the already fetched reserve data.
- Rounding errors can cause matching of dust. This could be avoided by using fractions to store principal values: Instead of dividing token amounts by an index and later multiplying again by an index (division before multiplication), principal values could be stored as `(uint128, uint128)` tuples of the base value and the index at that time. This change requires careful handling of `uint128` casts though.
- Some checks can be performed earlier in the code, saving callers some gas on reverting transactions:
    - `_borrowAllowed` in `EntryPositionsManager.borrowLogic`.
    - Maximum number of markets check in `MorphoGovernance.createMarket`.
- Unnecessary computations:
    - `ExitPositionsManager.withdrawLogic` does not have to check if the user is supplying. Instead, it could revert on `toWithdraw == 0`.

    - `ExitPositionsManager._safeWithdrawLogic` potentially calls `_updateSupplierInDS` 2 times with no overlap.

- Changes in the Heap data structures that result in one account being removed and another account being updated could be performed with a `replace` action instead of `pop` + `push`.
- Tighter packing of storage variables is possible (careful casting is necessary though).
    - `p2pSupplyIndex` and `p2pBorrowIndex` could be packed as `uint128` into a single struct, since most of the time, both values are read from the storage together.
    - The fields of the structs `SupplyBalance` and `BorrowBalance` could be reduced to `uint128`.
- Many variables in `MorphoStorage` (e.g. `entryPositionsManager`) could be defined as `immutable`. Since the `Morpho` contract is `Upgradeable`, the values can be changed by updating the proxy implementation.
- In `ExitPositionsManager._getUserHealthFactor` (and other functions with similar use-case), each asset price is individually fetched with `oracle.getAssetPrice`. Since the Aave oracle exposes a function to fetch multiple asset prices at once, some external calls can be saved by using `oracle.getAssetsPrices`.

Code partially corrected:

- **Corrected:** Underlying token addresses are now saved in the `market` storage variable.
- **Partially corrected:** Redundant storage reads have been improved on some occasions but still happen in various places.
- **Not corrected:** The mentioned examples have not been updated to reduce storage writes.
- **Not corrected:** The first example is obsolete because of another change, the other examples have not been addressed.
- **Not corrected:** The suggested change has not been implemented.
- **Partially corrected:** `createMarket` now checks for the maximum number of markets in the beginning of the function.
- **Corrected:** The mentioned redundant computations have been removed.
- **Not corrected:** `pop` + `push` is still used.
- **Not corrected:** Variables are not packed more tightly.
- **Not corrected:** No variables have been changed to `immutable`.
- **Not corrected:** `oracle.getAssetPrices` is not used.

5.5 Missing Sanity Checks

Design Low Version 1 Code Partially Corrected

- `Morpho.createMarket` does not check if the `_underlyingTokenAddress` is the 0-address.
- Multiple Governance setters do not check if address parameters are the 0-address.
- The initializer of `MorphoGovernance` does not check if `maxSortedUsers` is zero. However, the check is applied in `setMaxSortedUsers`.
- The `ExitPositionsManager.liquidateLogic` can be called with an `_amount` of `0`, while this is not possible in other entry points.

Code partially corrected:

- **Corrected:** Morpho.createMarket now checks if _underlyingToken is the 0-address.
- **Corrected:** Setters now check for the 0-address except if the respective field can be intentionally set to the 0-address.
- **Corrected:** The MorphoGovernance initializer now checks if maxSortedUsers is zero.
- **Not corrected:** ExitPositionsManager.liquidateLogic can still be called with an _amount of 0.

5.6 Rewards Can Be Withdrawn by Admins

Design Low Version 1 Risk Accepted

Morpho.claimRewards transfers accrued rewards of Morpho (Aave v3)'s whole position from Aave when a user claims their share of the rewards. This means that some tokens may now be owned by the Morpho contract, which can later be claimed by other users.

If one of the reward tokens is however an active market on Morpho (Aave v3), the tokens are claimable by the contract admin. In this case, both fees and user rewards are mixed together and the contract admin could accidentally mistake all of the tokens for fees and withdraw them.

This would result in rewards not being claimable by all users that are entitled to them.

Risk accepted:

Morpho Labs accepts the risk stating that the admin (or DAO) is not advised to withdraw fees when there is a running rewards program where the reward token is equal to one of the market tokens.

5.7 Variable Shadowing

Design Low Version 1 Risk Accepted

- In InterestRateManager.updateIndexes and MorphoGovernance.createMarket, the state variable poolIndexes is shadowed by a local variable.
- In MorphoUtils.isMarketCreatedAndNotPaused and isMarketCreatedAndNotPausedNorPartiallyPaused, the state variable marketStatus is shadowed by a local variable.

Risk accepted:

Morpho Labs accepts the risk. Furthermore, additional storage variables are shadowed in some functions now.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- Unadapted amountToLiquidate Code Corrected

Low-Severity Findings	16

- Deprecated Market Close Factor Less Than 100% Code Corrected
- Pause Functions Can Be Used on Any Token Address Code Corrected
- Free Borrowing of Small Amounts Possible Code Corrected
- Function Can Be Restricted to Pure Code Corrected
- Incorrect and Missing Specs Code Corrected
- Limited Liquidation Amount Code Corrected
- MorphoToken Not Safely Transferred Code Corrected
- P2pBorrowDelta Always Zero Code Corrected
- Potentially Different RewardsController Address Code Corrected
- Redundant Code Code Corrected
- Similar Code Abstraction Code Corrected
- Unused Imports / Errors Code Corrected
- Use of Deprecated Function Code Corrected
- Withdrawal Denial of Service Code Corrected
- Withdrawals Do Not Check Oracle Health Code Corrected
- Wrong Event Data Code Corrected

6.1 Unadapted amountToLiquidate

Design Medium Version 1 Code Corrected

In `liquidateLogic`, `amountToLiquidate` is computed before `amountToSeize` is capped. However, `amountToLiquidate` is not adapted to the capped `amountToSeize`, which may cause the liquidator to repay more than the value of the collateral they obtain.

Code corrected:

If `amountToSeize` exceeds the amount of the liquidated user's collateral balance of the requested token, `amountToLiquidate` is adjusted as follows:

```
amountToLiquidate = ((collateralBalance * collateralPrice * vars.borrowedTokenUnit) /
            (borrowedTokenPrice * vars.collateralTokenUnit))
        .percentDiv(vars.liquidationBonus);
```

6.2 Deprecated Market Close Factor Less Than 100%

Design Low Version 3 Code Corrected

Markets can be set to `deprecated` state in order to incentivize users to close their borrow positions in that market. Deprecated assets can be liquidated from accounts even if liquidations of such accounts would not be possible otherwise. Morpho (Aave v3) currently allows only 50% of a user's borrow position to be liquidated this way. As liquidations of deprecated assets bypass the `_liquidationAllowed` function, multiple consecutive liquidations of the same position are possible but a complete liquidation remains impracticable. A close factor of less than 100% is therefore not sensible in this case.

Code corrected:

Deprecated assets can now be fully liquidated.

6.3 Pause Functions Can Be Used on Any Token Address

Design Low Version 3 Code Corrected

Several pause function (e.g., `MorphoGovernance.setIsSupplyPaused`) do not check if the address of the given pool token is actually a valid market. If a non-valid market address is paused and that market is later created, the market might instantly be paused.

Code corrected:

All pause functions now check if the given pool token address belongs to a valid market.

6.4 Free Borrowing of Small Amounts Possible

Correctness Low Version 1 Code Corrected

Certain circumstances allow for the borrowing of very small amounts of tokens without supplying collateral beforehand:

`EntryPositionsManager._borrowAllowed` computes the values of the supplied and borrowed tokens in a base currency and checks whether an additional borrowed amount would result in the account being underwater.

For an account with 0 supplied and borrowed balances, the following computation determines if a borrow is allowed:

```
liquidityData.debtValue +=
    (_borrowedAmount * assetData.underlyingPrice) /
    assetData.tokenUnit;
```

Depending on the decimals of the token and the decimals of the price oracle, a small `_borrowedAmount` might result in integer division that rounds to 0. This would satisfy the final check and allow the borrowing of the given amount:

```
liquidityData.debtValue <= liquidityData.maxLoanToValue
```

Morpho (Aave v3) uses the oracles of the underlying AAVE pool, which in turn uses **Chainlink** price feeds. On ETH Mainnet, AAVE uses Chainlink price feeds in ETH base currency, which have 18 decimals (this is true for AAVE v2. AAVE v3 is not yet live on Mainnet at the time of this writing). On other chains (e.g. Optimism), AAVE uses feeds with USD base currency, which only have 8 decimals. In this case, many tokens become susceptible to this problem.

Since the claimable amounts are significantly lower than the amount of gas that has to be paid, the bug is of very low severity.

Code corrected:

The division by `tokenUnit` is now performed using the function `Math.divUp`. This function adds 1 wei to the debt if `(_borrowedAmount * underlyingPrice) % tokenUnit != 0`. Therefore, borrowing small amounts of tokens without sufficient collateral is not possible anymore.

6.5 Function Can Be Restricted to Pure

Design Low Version 1 Code Corrected

The function `RewardsManager._getRewards` can be restricted to `pure` as it does not read from the storage.

Code corrected:

`_getRewards` is now marked as `pure`.

6.6 Incorrect and Missing Specs

Design Low Version 1 Code Corrected

- In `MorphoGovernance`, doc comments wrongly specify a `_poolTokenAddress` parameter for the `MarketCreated` event.
- Function comments in `MatchingEngine.sol` should mention `Aave` instead of `Compound`.
- All fields of struct `Types.Delta` are expressed in underlying decimals instead of in `WAD` as claimed in the specs.
- Some parameters are missing in the specs of `RewardsManager._updateRewardData`.

Code corrected:

- **Corrected:** The `MarketCreated` event is now correctly documented.
- **Corrected:** The correct protocol name has now been added to all comments in `MatchingEngine`.
- **Corrected:** The correct decimal types are now documented for all `Types.Delta` fields.

- **Corrected:** `RewardsManager._updateRewardData` parameters are now correctly documented.

6.7 Limited Liquidation Amount

Design Low Version 1 Code Corrected

Liquidations in Morpho (Aave v3) are only allowed up to a maximum of 50% of the user's borrowed assets. This is true even when the health factor of the user is below 95%. This is contrary to the implementation of the underlying Aave pool, which allows for a liquidation of the whole user position when the user's health factor drops below 95%.

This behavior can increase the risk of Morpho's position on Aave becoming liquidatable (for example because liquidation bots on Morpho are not working efficiently).

Code corrected:

User positions with a health factor below 95% can now be liquidated completely. `ExitPositionsManager._liquidationAllowed` returns the respective liquidation close factor.

6.8 MorphoToken Not Safely Transferred

Design Low Version 1 Code Corrected

`IncentivesVault.tradeRewardTokensForMorphoTokens` transfers MORPHO tokens without checking a possible return. It is advised to use `SafeTransferLib.safeTransfer` in this case.

This might not be necessary depending on the implementation of the MORPHO token. At the time of this writing, no such contract is known to us. Therefore, we are unable to verify if the use of `transfer` is safe in this case.

Code corrected:

`tradeRewardTokensForMorphoTokens` now uses `safeTransfer` to transfer MORPHO tokens.

6.9 P2pBorrowDelta Always Zero

Design Low Version 1 Code Corrected

When repaying the fee in `ExitPositionsManager._safeRepayLogic`, `delta.p2pBorrowDelta` has already been reduced to `0` at this point and could be safely removed from the equation.

Code corrected:

`_safeRepayLogic` does not take `delta.p2pBorrowDelta` into account anymore.

6.10 Potentially Different RewardsController Address

Design Low Version 1 Code Corrected

The Morpho and RewardsManager contracts may have different RewardsController addresses as there is no synchronization between them at initialization.

Code corrected:

RewardsManager does not store the RewardsController address anymore. Instead, the address is passed to its functions as an argument.

6.11 Redundant Code

Design Low Version 1 Code Corrected

- In EntryPositionManager the second if check is redundant as shown below.

```
if (toWithdraw > 0) {
    uint256 toAddInP2P = toWithdraw.rayDiv(p2pBorrowIndex[_poolTokenAddress]); // In peer-to-peer unit.

    deltas[_poolTokenAddress].p2pBorrowAmount += toAddInP2P;
    borrowBalanceInOf[_poolTokenAddress][msg.sender].inP2P += toAddInP2P;
    emit P2PAmountsUpdated(_poolTokenAddress, delta.p2pSupplyAmount, delta.p2pBorrowAmount);

    if (toWithdraw > 0) _withdrawFromPool(underlyingToken, _poolTokenAddress, toWithdraw); // Reverts on error.
}
```

- In EntryPositionManager.sol, the function _borrowAllowed does not need to check if _amount == 0, because this is already checked at the beginning of borrowLogic.
- In liquidateLogic, the check _isBorrowingAny(_borrower) is redundant, because it is already checked at the beginning by _isBorrowing(_borrower, _poolTokenBorrowedAddress).

Code corrected:

The redundant code parts have been removed / are not relevant anymore.

6.12 Similar Code Abstraction

Design Low Version 1 Code Corrected

The functions ExitPositionsManager._getUserHealthFactor and EntryPositionsManager._borrowAllowed share a similar code base that should be abstracted away to avoid maintenance problems.

Code corrected:

The common logic of ExitPositionsManager._getUserHealthFactor and EntryPositionsManager._borrowAllowed has been abstracted into the function MorphoUtils._liquidityData.

6.13 Unused Imports / Errors

Design Low Version 1 Code Corrected

MorphoGovernance defines the AmountIsZero error, but it is never used in the inheritance hierarchy of the contract.

Code corrected:

The AmountIsZero error has been removed from MorphoGovernance.

6.14 Use of Deprecated Function

Design Low Version 1 Code Corrected

PositionsManagerUtils.supplyToPool calls the pool.deposit function which is deprecated in Aave.

Code corrected:

supplyToPool now calls pool.supply on Aave instead.

6.15 Withdrawal Denial of Service

Correctness Low Version 1 Code Corrected

ExitPositionsManager.withdrawLogic calls _getUserHealthFactor if the user is borrowing any tokens. If the user's borrow balance is small and if the called Aave oracle returns a number with lower decimals than the token's, then _getUserHealthFactor will revert on division by zero, preventing any withdrawals by the user.

This issue is related to Free borrowing of small amounts possible.

Code corrected:

The division by tokenUnit is now performed using the function Math.divUp. This function adds 1 wei to the debt if (_borrowedAmount * underlyingPrice) % tokenUnit != 0. Therefore, a division by zero even with small debts is not possible anymore.

6.16 Withdrawals Do Not Check Oracle Health

Design Low Version 1 Code Corrected

On withdrawals, Morpho (Aave v3) does not check for oracle health through the `PriceOracleSentinel` of AAVE. While AAVE does this the same way, there are still risks associated. Consider the following example:

- A tokens and B tokens are both worth exactly 1 USD.
- A user supplies 500 A tokens and borrows 200 B tokens.
- The user can withdraw up to 200 A tokens and still maintain a good health factor.
- Now the price of A tokens rapidly changes to 0.2 USD, but the price oracle for A tokens has not updated for a few days and still shows 1 USD / A token.
- The user is now still able to withdraw 200 A tokens while in reality, his position is already under water.

As the recent debacle with Chainlink price feeds of LUNA has shown, oracles that are not updating prices in a timely manner can become very problematic for lending protocols. It is therefore advised to check the health of such oracles.

Unfortunately, at the time of this writing, AAVE has not deployed a `PriceOracleSentinel` so the problem persists also for liquidations and borrowing until AAVE deploys these mechanisms.

Code corrected:

`ExitPositionsManager._withdrawAllowed` now checks the oracle health by calling `priceOracleSentinel.isBorrowAllowed()`.

6.17 Wrong Event Data

Correctness Low Version 1 Code Corrected

The following events contain wrong data:

- `ExitPositionsManager._safeWithdrawLogic` emits the event `P2PBorrowDeltaUpdated` with `delta.p2pBorrowAmount` instead of `delta.p2pBorrowDelta`.
- `ExitPositionsManager._safeRepayLogic` emits the event `P2PSupplyDeltaUpdated` with `delta.p2pBorrowDelta` instead of `delta.p2pSupplyDelta`.

Code corrected:

The mentioned events are now emitted using the correct parameters.

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Accidental Ownership Transfers

Note Version 1

Morpho (Aave v3) contracts use OpenZeppelin's `Ownable` contract to store ownership. `Ownable` employs a single-step ownership transfer. Accidental transfers to the wrong address will lock out the owner indefinitely. For this reason, special care has to be taken when updating the ownership of the contracts.

7.2 Delta Reduced When P2P Is Disabled

Note Version 1

If `p2pDisabled` is set to true, the peer-to-peer delta is reduced instead of borrowers / suppliers unmatched.

7.3 IncentivesVault Security Relies on Oracle Implementation

Note Version 1

The Security of `IncentivesVault.tradeRewardTokensForMorphoTokens` relies on the implementation of the oracle that is set to calculate the value of the given rewards. Since there is no implementation available at the time of this writing, we cannot attest if the use of this function is secure.

Morpho Labs aims to implement a TWAP oracle based on Uniswap v3 in the future.

7.4 Lack of Balance Functions

Note Version 1

The `Morpho` contract does not expose view functions for user balances.

7.5 Liquidation Risk on Aave

Note Version 1

If Morpho's position on Aave is liquidated, the unmatched accounting of Morpho and Aave would break Morpho's availability and possibly lock users' funds. Besides, fully pausing Morpho would increase the liquidation risk on Aave. Efficient arbitrage bots are required to run on Morpho so that the underlying position on Aave does not become liquidatable.

7.6 Liquidations May Affect Rates of P2P Users

Note Version 1

ExitPositionsManager.liquidateLogic calls repayLogic and withdrawLogic with 0 gas for matching. This can lead to worse rates for P2P users as supply / borrow delta can be increased by these operations.

7.7 No Delay Mechanism for Parameter Updates

Note Version 1

There is no delay mechanism for the updates of parameters to take into effect. Users who are not satisfied with the upcoming updates would not have time to leave the market.

7.8 Potentially Exceeding maxGasForMatching

Note Version 1

As shown below, the matching functions potentially use slightly more gas than the users' given _maxGasForMatching.

```
while (
        remainingToMatch > 0 &&
        (firstPoolSupplier = suppliersOnPool[_poolTokenAddress].getHead()) != address(0)
) {
    unchecked {
        if (gasLeftAtTheBeginning - gasleft() >= _maxGasForMatching) break;
    }
    ...
}
```

7.9 Unsupported Tokens

Note Version 1

The following tokens can not be used in Morpho Markets without repercussions:

- Aave siloed assets.
- Aave isolated assets.
- Tokens with high decimals (e.g. 27) because the amountToSeize calculation in EntryPositionsManager.liquidateLogic might overflow on realistic token amount values.
- Tokens that charge a transfer fee (e.g. STA, PAXG).
- Rebasing tokens.

Addendum:

Aave isolated assets can now be used as long as they are not used as collateral on the underlying pool.

7.10 Year 2106 Problem for Uint32 Timestamps

Note Version 1

Timestamps are written to storage which could impose problems on the storage layout in the year 2106.

7.11 safeTransferFrom Does Not Revert on Calls to an EOA

Note Version 1

Morpho (Aave v3) uses Rari Capital's `SafeTransferLib` to support tokens that revert on transfers as well as tokens that return a boolean value. The functions, especially `safeTransferFrom`, however do not revert if the called token is not a contract. In this case, Morpho (Aave v3) contracts could be tricked into thinking that a token transfer from a user was successful when in fact nothing happened.

As of now, Morpho (Aave v3) is not affected by this behavior since all token addresses are directly taken from Aave which correctly checks for contracts in its `safeTransferFrom` function.

Future changes of the code should take this into account.