SPEARBIT

Morpho Security Review

Auditors

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Security Researcher

JayJonah8, Security Researcher

Datapunk, Junior Security Researcher

EBaizel, Junior Security Researcher

Report prepared by: Pablo Misirov

March 1, 2023

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Morpho is a lending pool optimizer. It improves the capital efficiency of positions on existing lending pools by seamlessly matching users peer-to-peer.

Morpho's rates stay between the supply rate and the borrow rate of the pool, reducing the interests paid by the borrowers while increasing the interests earned by the suppliers. It means that you are getting boosted peer-to-peer rates or, in the worst case scenario, the APY of the pool. Morpho also preserves the same experience, liquidity and parameters (collateral factors, oracles, . . .) as the underlying pool.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Morpho-v1 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 10 days in total, Morpho engaged with Spearbit to review the morpho-v1 protocol. In this period of time a total of **51** issues were found.

Summary

Project Name	Morpho
Repository	morpho-v1
Commit	5f39e0...1232
Type of Project	Lending and Borrowing, DeFi
Audit Timeline	Nov 21 - Dec 2
Fix revisions	Feb 13 - Feb 16

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	0	1
Medium Risk	14	5	9
Low Risk	4	1	3
Gas Optimizations	2	1	1
Informational	30	9	21
Total	**51**	**16**	**35**

5 Findings

5.1 High Risk

5.1.1 Liquidating Morpho's Aave position leads to state desync

Severity: High Risk

Context: ExitPositionsManager.sol#L239

Description: Morpho has a single position on Aave that encompasses all of Morpho's individual user positions that are on the pool. When this Aave Morpho position is liquidated the user position state tracked in Morpho desyncs from the actual Aave position. This leads to issues when users try to withdraw their collateral or repay their debt from Morpho. It's also possible to double-liquidate for a profit.

Example: There's a single borrower B1 on Morpho who is connected to the Aave pool.

B1 supplies 1 ETH and borrows 2500 DAI. This creates a position on Aave for Morpho The ETH price crashes and the position becomes liquidatable. A liquidator liquidates the position on Aave, earning the liquidation bonus. They repaid some debt and seized some collateral for profit. This repaid debt / removed collateral is not synced with Morpho. The user's supply and debt balance remain 1 ETH and 2500 DAI. The same user on Morpho can be liquidated again because Morpho uses the exact same liquidation parameters as Aave. The Morpho liquidation call again repays debt on the Aave position and withdraws collateral with a second liquidation bonus. The state remains desynced.

Recommendation: Liquidating the Morpho position should not break core functionality for Morpho users.

Morpho: We will not implement any "direct" fix inside the code.

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 A market could be deprecated but still prevent liquidators to liquidate borrowers if `isLiquidateBorrowPaused` is `true`

Severity: Medium Risk

Context: aave-v2/MorphoGovernance.sol#L358-L366, compound/MorphoGovernance.sol#L368-L376

Description: Currently, when a market must be deprecated, Morpho checks that borrowing has been paused before applying the new value for the flag.

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

The same check should be done in `isLiquidateBorrowPaused`, allowing the deprecation of a market only if `isLiquidateBorrowPaused == false` otherwise liquidators would not be able to liquidate borrowers on a deprecated market.

Recommendation: Prevent the deprecation of a market if the `isLiquidateBorrowPaused` flag is set to `true`.

Consider also checking the `isDeprecated` flag in the `setIsLiquidateBorrowPaused` to prevent pausing the liquidation if the market is deprecated. If Morpho implements the specific behavior should also be aware of the issue described in *"`setIsPausedForAllMarkets` bypass the check done in `setIsBorrowPaused` and allow resuming borrow on a deprecated market"*.

Morpho: We acknowledge this issue. The reason behind this is the following: given what @MathisGD said, if we want to be consistent we should prevent pausing the liquidation borrow on a deprecated asset. However, there might be an issue (we don't know) with the liquidation borrow and the operator would not be able to pause it. For this reason, we prefer to leave things as it is.

Spearbit: Acknowledged.

5.2.2 `setIsPausedForAllMarkets` bypass the check done in `setIsBorrowPaused` and allow resuming borrow on a deprecated market

Severity: Medium Risk

Context: aave-v2/MorphoGovernance.sol#L470, compound/MorphoGovernance.sol#L470

Description: The `MorphoGovernance` contract allow Morpho to set the `isBorrowPaused` to `false` only if the market is not deprecated.

```
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);
}
```

This check is not enforced by the `_setPauseStatus` function, called by `setIsPausedForAllMarkets` allowing Morpho to resume borrowing for deprecated market.

Test to reproduce the issue

```
// SPDX-License-Identifier: AGPL-3.0-only
pragma solidity ^0.8.0;

import "./setup/TestSetup.sol";

contract TestSpearbit is TestSetup {
    using WadRayMath for uint256;

    function testBorrowPauseCheckSkipped() public {
        // Deprecate a market
        morpho.setIsBorrowPaused(aDai, true);
        morpho.setIsDeprecated(aDai, true);
        checkPauseEquality(aDai, true, true);

        // you cannot resume the borrowing if the market is deprecated
        hevm.expectRevert(abi.encodeWithSignature("MarketIsDeprecated()"));
        morpho.setIsBorrowPaused(aDai, false);
        checkPauseEquality(aDai, true, true);

        // but this check is skipped if I call directly `setIsPausedForAllMarkets`
        morpho.setIsPausedForAllMarkets(false);

        // this should revert because
        // you cannot resume borrowing for a deprecated market
        checkPauseEquality(aDai, false, true);
    }

    function checkPauseEquality(
        address aToken,
        bool shouldBePaused,
        bool shouldBeDeprecated
    ) public {
        (
            bool isSupplyPaused,
            bool isBorrowPaused,
            bool isWithdrawPaused,
            bool isRepayPaused,
            bool isLiquidateCollateralPaused,
            bool isLiquidateBorrowPaused,
            bool isDeprecated
        ) = morpho.marketPauseStatus(aToken);

        assertEq(isBorrowPaused, shouldBePaused);
        assertEq(isDeprecated, shouldBeDeprecated);
    }

}
```

Recommendation: One possible solution is to follow the same approach implemented in the `aave-v3` codebase. The update of the `isBorrowPaused` is done only and only if the market is **not deprecated**.

Morpho: This issue has been fixed in PR 1642.

Spearbit: Verified.

5.2.3 User withdrawals can fail if Morpho position is close to liquidation

Severity: Medium Risk

Context: ExitPositionsManager.sol#L468

Description: When trying to withdraw funds from Morpho as a P2P supplier the last step of the withdrawal algorithm borrows an amount from the pool ("hard withdraw"). If the Morpho position on Aave's debt / collateral value is higher than the market's max LTV ratio but lower than the market's liquidation threshold, the borrow will fail and the position can also not be liquidated. The withdrawals could fail.

Recommendation: This seems hard to solve in the current system as it relies on the "hard withdraws" to always ensure enough liquidity for P2P suppliers. Consider ways to mitigate the impact of this problem.

Morpho: Since Morpho will first launch on Compound (where there is only Collateral Factor), we will not focus now on this particular issue.

Spearbit: Acknowledged.

5.2.4 P2P borrowers' rate can be reduced

Severity: Medium Risk

Original Context: (this area of the code has changed significantly since the initial audit, so maintaining the link to the original code base) MarketsManagerForAave.sol#L448

Context: aave-v2/InterestRatesManager.sol#L182 compound/InterestRatesManager.sol#L164

Description: Users on the pool currently earn a much worse rate than users with P2P credit lines. There's a queue for being connected P2P. As this queue could not be fully processed in a single transaction the protocol introduces the concept of a max iteration count and a borrower/supplier "delta" (c.f. yellow paper). This delta leads to a worse rate for existing P2P users. An attacker can force a delta to be introduced, leading to worse rates than before.

Example: Imagine some borrowers are matched P2P (earning a low borrow rate), and many are still on the pool and therefore in the pool queue (earning a worse borrow rate from Aave).

- An attacker supplies a huge amount, creating a P2P credit line for every borrower. (They can repeat this step several times if the max iterations limit is reached.)

- The attacker immediately withdraws the supplied amount again. The protocol now attempts to demote the borrowers and reconnect them to the pool. But the algorithm performs a "hard withdraw" as the last step if it reaches the max iteration limit, creating a *borrower delta*. These are funds borrowed from the pool (at a higher borrowing rate) that are still wrongly recorded to be in a P2P position for some borrowers. This increase in borrowing rate is socialized equally among all P2P borrowers. (reflected in an updated `p2pBorrowRate` as the `shareOfDelta` increased.)
- The initial P2P borrowers earn a worse rate than before. If the borrower delta is large, it's close to the on-pool rate.
- If an attacker-controlled borrower account was newly matched P2P and *not* properly reconnected to the pool (in the "demote borrowers" step of the algorithm), they will earn a better P2P rate than the on-pool rate they earned before.

Recommendation: Consider mitigations for single-transaction flash supply & withdraw attacks.

Morpho: We acknowledge the risk and won't be fixing this issue, as we might want to refactor the entire queue system at some point.

Spearbit: Acknowledged.

5.2.5 Frontrunners can exploit system by not allowing head of DLL to match in P2P

Severity: Medium Risk

Context: MatchingEngine.sol

Description: For a given asset `x`, liquidity is supplied on the pool since there are not enough borrowers. `suppliersOnPool` head: `0xa` with 1000 units of `x`

whenever there is a new transaction in the mempool to borrow 100 units of `x`,

- Frontrunner supplies 1001 units of `x` and is supplied on pool.
- `updateSuppliers` will put the frontrunner on the head (assuming very high gas is supplied).
- Borrower's transaction lands and is matched 100 units of `x` with a frontrunner in p2p.
- Frontrunner withdraws the remaining 901 left which was on the underlying pool.

Favorable conditions for an attack:

- Relatively fewer gas fees & relatively high block gas limit.
- `insertSorted` is able to traverse to head within block gas limit (i.e length of DLL).

Since this is a non-atomic sandwich, the frontrunner needs excessive capital for a block's time period.

Recommendation: Consider mitigations for frontrunning sandwich attacks.

Morpho: We acknowledge this issue and we are currently searching for better matching engine mechanisms. Though, as we must prevent the protocol from DDOs attacks a classic FIFO is not possible.

We'll keep the matching engine like it is as the result of the front-running attack you mentioned is similar to a whale with huge capital which would be at the head of the list.

Spearbit: Acknowledged.

5.2.6 Differences between Morpho and Compound `borrow` validation logic

Severity: Medium Risk

Context: compound/PositionsManager.sol#L336-L344

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between them;

- Compound has a mechanism to prevent borrows if the new borrowed amount would go above the current `borrowCaps[cToken]` threshold. Morpho does not check this threshold and could allow users to borrow on the P2P side (avoiding the revert because it would not trigger the underlying compound borrow action). Morpho should anyway monitor the `borrowCaps` of the market because it could make `increaseP2PDeltasLogic` and `_unsafeWithdrawLogic` reverts.
- Both Morpho and Compound do not check if a market is in "deprecated" state. This means that as soon as a user borrows some tokens, he/she can be instantly liquidated by another user.
  - If the flag is true on Compound, the Morpho User can be liquidated directly on compound.
  - If the flag is true on Morpho, the borrower can be liquidated on Morpho.
- Morpho does not check if `borrowGuardianPaused[cToken]` on Compound, a user could be able to borrow in P2P while the `cToken` market has borrow paused.

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Compound"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.2.7 Users can continue to borrow from a deprecated market

Severity: Medium Risk

Context: aave-v2/MorphoGovernance.sol#L395 compound/MorphoGovernance.sol#L372

Description: When a market is being marked as deprecated, there is no verification that the borrow for that market has already been disabled. This means a user could borrow from this market and immediately be eligible to be liquidated.

Recommendation: A couple of options:

- Add a `require` or `modifier` to ensure `borrow` has been disabled, and `revert` if not.
- Disable `borrow` as part of deprecating the market.

Morpho: Fixed in PR 1551.

Spearbit: Verified. After the PR change, to be able to deprecate a market, Morpho must pause the borrowing state; otherwise, the transaction will revert. When both the borrow state is paused and the market is deprecated, if Morpho wants to "reset" those values (borrow not paused, and the market is not deprecated), Morpho must "un-deprecate" it and only then "un-pause" it.

Note: Morpho should check that all the markets created are both **not deprecated** and **not borrow-paused** before deploying the PR to be sure to not enter a case where the new checks would not work or would prevent resetting the flags because the system is in an inconsistent state.

5.2.8 ERC20 with transfer's fee are not handled by `*PositionManager`

Severity: Medium Risk

Context: PositionsManager.sol, EntryPositionsManager.sol, ExitPositionsManager.sol

Description: Some ERC20 tokens could have fees attached to the `transfer` event, while others could enable them in the future (see `USDT`, `USDC`).

The current implementation of both `PositionManager` (for Aave and Compound) is not taking into consideration these types of ERC20 tokens. While Aave seems not to take into consideration this behavior (see LendingPool.sol), Compound, on the other hand, is explicitly handling it inside the doTransferIn function. Morpho is taking for granted that the amount specified by the user will be the amount transferred to the contract's balance, while in reality, the contract will receive less.

In `supplyLogic`, for example, Morpho will account for the user's p2p/pool balance for the full `amount` but will repay/supply to the pool less than the `amount` accounted for.

Recommendation: Consider updating the `*PositionManager` logic to track the real amount of token that has been sent by the user after the transfer (difference in before and after balance), but also the amount of tokens that have been supplied/borrowed/withdrawn/... given that Morpho itself is doing a second `transfer/transferFrom` to/from the Aave/Compound protocol.

Morpho: We updated the asset listing checklist. However, given the small likelihood for USDC and USDT to turn on fees, we decided not to implement the recommendations.

Spearbit: Verified the checklist, and Acknowledged the recommendations will not be implemented.

5.2.9 Cannot liquidate Morpho users if no liquidity on the pool

Severity: Medium Risk

Context: aave-v2/ExitPositionsManager.sol#L277

Description: Morpho implements liquidations by repaying the borrowed asset and then withdrawing the collateral asset from the underlying protocol (Aave / Compound). If there is no liquidity in the collateral asset pool the liquidation will fail. Morpho could incur bad debt as they cannot liquidate the user. The liquidation mechanisms of Aave and Compound work differently: They allow the liquidator to seize the debtorsTokens/cTokens which can later be withdrawn for the underlying token once there is enough liquidity in the pool.

Technically, an attacker could even force no liquidity on the pool by frontrunning liquidations by borrowing the entire pool amount - preventing them from being liquidated on Morpho. However, this would require significant capital as collateral in most cases.

Recommendation: Think about adding a similar feature where liquidators can seize aTokens/cTokens instead of withdrawing underlying tokens from the pool. The aTokens/cTokens of all pool users are already in the Morpho contract and thus in Morpho's control. Note that this would only work with `onPool` balances but not with `inP2P` balances as these don't mint aTokens/cTokens.

Morpho: As it requires large changes in the liquidation process, we decided not to implement this recommendation on the current codebase.

Spearbit: Acknowledged.

5.2.10 Supplying and borrowing can recreate p2p credit lines even if p2p is disabled

Severity: Medium Risk

Context: aave-v2/EntryPositionsManager.sol#L117, aave-v2/EntryPositionsManager.sol#L215, compound/PositionsManager.sol#L258, compound/PositionsManager.sol#L354

Description: When supplying/borrowing the algorithm tries to reduce the deltas `p2pBorrowDelta/p2pSupplyDelta` by moving borrowers/suppliers back to P2P. It is not checked if P2P is enabled. This has some consequences related to when governance disables P2P and wants to put users and liquidity back on the pool through `increaseDelta` calls. The users could enter P2P again by supplying and borrowing.

Recommendation: Disable matching the initial delta-matching step in `supply` and `borrow` if P2P is disabled.

> *The reason why this only needs to be done for `supply` and `borrow` and not for `repay` and `withdraw` is that for `repay` and `withdraw`, while we're also reducing the delta, we're not actually creating new p2p credit lines (p2pAmount also decreases, so the diff is zero). It can be seen as if we're unmatching our own p2p balance, reducing the delta, shifting our p2p balance to on pool, and then withdrawing from the pool.*

Morpho: Fixed in PR 1453.

Spearbit: Verified.

5.2.11 In Compound implementation, P2P indexes can be stale

Severity: Medium Risk

Context: MorphoUtils.sol#L119-L156, PositionsManager.sol#L344, PositionsManager.sol#L447, PositionsManager.sol#L502-L505

Description: The current implementation of `MorphoUtils._isLiquidatable` loops through all of the tokens in which the user has supplied to/borrowed from. The scope of the function is to check whether the user can be liquidated or not by verifying that `debtValue > maxDebtValue`.

Resolving *"Compound liquidity computation uses outdated cached borrowIndex"* implies that the Compound borrow index used is always up-to-date but the P2P issues associated with the token could still be out of date if the market has not been used recently, and the underlying Compound indexes (on which the P2P index is based) has changed a lot.

As a consequence, all the functions that rely on `_isLiquidatable` (liquidate, withdraw, borrow) could return a wrong result if the majority of the user's balance is on the P2P balance (the problem is even more aggravated without resolving *"Compound liquidity computation uses outdated cached borrowIndex"*.

Let's say, for example:

- Alice supplies ETH in pool
- Alice supplies BAT in P2P
- Alice borrows some DAI

At some point in time the ETH value goes down, but the interest rate of BAT goes up. If the P2P index of BAT had been correctly up-to-date, Alice would have been still solvent, but she gets liquidated by Bob who calls `liquidate(alice, ETH, DAI)`

Even by fixing *"Compound liquidity computation uses outdated cached borrowIndex"* Alice would still be liquidated because her entire collateral is on P2P and not in the pool.

Recommendation: Consider following the same approach implemented in the Morpho-Aave implementation inside MorphoUtils._liquidityData that will always be the token that the user has supplied/borrowed.

Unlike Aave, Morpho's Compound implementation does not have a maximum hard-cap limit, which means that the `_isLiquidatable` loop could possibly revert because of an Out of Gas exception.

Ultimately, Morpho should always remember to always call `updateP2PIndexes` (for both Aave and Compound) before any logic inside the `*PositionsManager` (both Aave and Compound).

Morpho: Because liquidators are updating indexed to be able to perform liquidations and because it would drastically increase the gas cost, we decided not to implement this recommendation.

Spearbit: Acknowledged.

5.2.12 Turning off an asset as collateral on Morpho-Aave still allows seizing of that collateral on Morpho and leads to liquidations

Severity: Medium Risk

Context: aave-v2/MorphoGovernance.sol#L407, aave-v2/MorphoUtils.sol#L285

Description: The Morpho Aave deployment can set the asset to not be used as collateral for Aave's Morpho contract position. On Aave, this prevents liquidators from seizing this asset as collateral.

1. However, this prevention does not extend to users on Morpho as Morpho has not implemented this check. Liquidations are performed through a `repay` & `withdraw` combination and `withdraw`ing the asset on Aave is still allowed.
2. When turning off the asset as collateral, the single Morpho contract position on Aave might still be over-collateralized, but some users on Morpho suddenly lose this asset as collateral (LTV becomes 0) and will be liquidated.

Recommendation: The feature does not work well with the current version of the Morpho Aave contracts. It must be enabled right from the beginning and may not be set later when users are already borrowing against the asset as collateral on Morpho. Clarify when this feature is supposed to be used, taking into consideration the mentioned issues. Reconsider if it's required.

Morpho: Fixed in PR 1542.

Spearbit: Verified.

5.2.13 `claimToTreasury(COMP)` steals users' COMP rewards

Severity: Medium Risk

Context: compound/MorphoGovernance.sol#L414

Description: The `claimToTreasury` function can send a market's underlying tokens that have been accumulated in the contract to the treasury. This is intended to be used for the reserve amounts that accumulate in the contract from P2P matches. However, Compound also pays out rewards in COMP and COMP is a valid Compound market. Sending the COMP reserves will also send the COMP rewards. This is especially bad as anyone can claim COMP rewards on the behalf of Morpho at any time and the rewards will be sent to the contract. An attacker could even frontrun a `claimToTreasury(cCOMP)` call with a `Comptroller.claimComp(morpho, [cComp])` call to sabotage the reward system. Users won't be able to claim their rewards.

Recommendation: If Morpho wants to support the COMP market, consider separating the COMP reserve from the COMP rewards.

Morpho: Given the changes to do and the small likelihood to set a reserve factor for the COMP asset and the awareness on our side about this, we decided not to implement it.

Spearbit: Acknowledged.

5.2.14 Compound liquidity computation uses outdated cached borrowIndex

Severity: Medium Risk

Context: compound/MorphoUtils.sol#L211

Description: The `_isLiquidatable` iterates over all user-entered markets and calls `_getUserLiquidityDataForAsset(poolToken) -> _getUserBorrowBalanceInOf(poolToken)`. However, it only updates the indexes of markets that correspond to the borrow and collateral assets. The `_getUserBorrowBalanceInOf` function computes the underlying pool amount of the user as `userBorrowBalance.onPool.mul(lastPoolIndexes[_poolToken].lastBorrowPoolIndex);`. Note that `lastPoolIndexes[_poolToken].lastBorrowPoolIndex` is a value that was cached by Morpho and it can be outdated if there has not been a user-interaction with that market for a long time.

The liquidation does not match Compound's liquidation anymore and users might not be liquidated on Morpho that could be liquidated on Compound. Liquidators would first need to trigger updates to Morpho's internal borrow indexes.

Recommendation: To match Compound's liquidation procedure, consider using Compound's `borrowIndex` which might have been updated after Morpho updated its own internal indexes.

```
function _getUserBorrowBalanceInOf(address _poolToken, address _user)
    internal
    view
    returns (uint256)
{
    Types.BorrowBalance memory userBorrowBalance = borrowBalanceInOf[_poolToken][_user];
    return
        userBorrowBalance.inP2P.mul(p2pBorrowIndex[_poolToken]) +
-       userBorrowBalance.onPool.mul(lastPoolIndexes[_poolToken].lastBorrowPoolIndex);
+       userBorrowBalance.onPool.mul(ICToken(_poolToken).borrowIndex());
}
```

Morpho: Fixed in PR 1558.

Spearbit: Verified.

5.3 Low Risk

5.3.1 `HeapOrdering.getNext` returns the root node for nodes not in the list

Severity: Low Risk

Context: HeapOrdering.sol#L328

Description: If an id does not exist in the `HeapOrdering` the `getNext()` function will return the root node

```
uint256 rank = _heap.ranks[_id]; // @audit returns 0 as rank. rank + 1 will be the root
if (rank < _heap.accounts.length) return getAccount(_heap, rank + 1).id;
else return address(0);
```

Recommendation: Consider returning the zero address if the `rank` variable is zero (the `_id` was not found).

Morpho: This issue has been fixed in PR 107 and PR 1627.

Spearbit: Verified.

5.3.2 Heap only supports balances up to `type(uint96).max`

Severity: Low Risk

Context: HeapOrdering.sol#L9

Description: The current heap implementation packs an `address` and the balance into a single storage slot which restricts the balance to the `uint96` type with a max value of `~7.9e28`. If a token has 18 decimals, the largest balance that can be stored will be `7.9e10`. This could lead to problems with a token of low value, for example, if `1.0` tokens are worth `0.0001$`, a user could only store `7_900_000$`.

Recommendation: The Aave markets currently don't list a token of such low value. Check the token values before listing an Aave market on Morpho, or consider increasing the balance.

Morpho: We updated the asset listing checklist, PR 6.

Spearbit: Acknowledged.

5.3.3 Delta leads to incorrect reward distributions

Severity: Low Risk

Context: aave-v2/File.sol#L123

Description: Delta describes the amount that is on the pool but still wrongly tracked as inP2P for some users. There are users that do not have their P2P balance updated to an equivalent pool balance and therefore do not earn rewards. There is now a mismatch of this delta between the pool balance that earns a reward and the sum of pool balances that are tracked in the reward manager to earn that reward. The increase in delta directly leads to an increase in rewards for all other users on the pool.

Recommendation: In a future version, think about distributing the share of the delta on the balance that earns rewards (`delta / (onPool + delta)`) to all P2P suppliers.

Morpho: These are quite involved and difficult tasks that would require a lot of changes. For these reasons, we decided not to implement the recommendation.

Spearbit: Acknowledged.

5.3.4 When adding a new rewards manager, users already on the pool won't be earning rewards

Severity: Low Risk

Context: aave-v2/MatchingEngine.sol#L315, compound/MatchingEngine.sol#L347

Description: When setting a new rewards manager, existing users that are already on the pool are not tracked and won't be earning rewards.

Recommendation: There's currently no efficient way to fix this besides initializing the new reward manager with all users who are already on the pool. Users with large pool supplies can resupply / reborrow a tiny amount to the pool to be registered in the new rewards manager.

Morpho: It was like this when Aave still had rewards. We have removed it for Aave in PR 1545 and PR 1538. For Compound, we acknowledge this issue. Users will be warned in any case before a rewards manager is set to 0.

Spearbit: Verified the Aave rewards removal, and Acknowledged the approach for Compound.

5.4 Gas Optimization

5.4.1 `liquidationThreshold` computation can be moved for gas efficiency

Severity: Gas Optimization

Context: aave-v2/MorphoUtils.sol#L320-L323

Description: The `vars.liquidationThreshold` computation is only relevant if the user is supplying this asset. Therefore, it can be moved to the `if (_isSupplying(vars.userMarkets, vars.borrowMask))` branch.

Recommendation: Consider changing the code

```
// Cache current asset collateral value.
uint256 assetCollateralValue;
if (_isSupplying(vars.userMarkets, vars.borrowMask)) {
    assetCollateralValue = _collateralValue(
        vars.poolToken,
        _user,
        vars.underlyingPrice,
        assetData.tokenUnit
    );
    values.collateral += assetCollateralValue;
    // Calculate LTV for borrow.
    values.maxDebt += assetCollateralValue.percentMul(assetData.ltv);

+   // Update LT variable for withdraw.
+   if (assetCollateralValue > 0)
+       values.liquidationThreshold += assetCollateralValue.percentMul(
+           assetData.liquidationThreshold
+       );
}

// Update debt variable for borrowed token.
if (_poolToken == vars.poolToken && _amountBorrowed > 0)
    values.debt += (_amountBorrowed * vars.underlyingPrice).divUp(assetData.tokenUnit);

- // Update LT variable for withdraw.
- if (assetCollateralValue > 0)
-     values.liquidationThreshold += assetCollateralValue.percentMul(
-         assetData.liquidationThreshold
-     );
```

Morpho: Fixed in PR 1544.

Spearbit: Verified. However, there is a further optimization that can be made. Sample code is provided in *"`liquidationThreshold` computation can be moved for gas efficiency"*.

5.4.2 Add max approvals to markets upon market creation

Severity: Gas Optimization

Context: aave-v2/File.sol#L123

Description: Approvals to the Compound markets are set on each `supplyToPool` function call.

Recommendation: Consider adding a single max approval of `type(uint256).max` once upon market creation in `MorphoGovernance.createMarket` to save gas, and remove the other approvals. The Aave-v2 contracts are already doing this.

Morpho: Because it adds complexity to the upgrade process for a too small upside, we decided not to implement it.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 isP2PDisabled flag is not updated by setIsPausedForAllMarkets

Severity: Informational

Context: aave-v2/MorphoGovernance.sol#L466-L482, compound/MorphoGovernance.sol#L466-L482

Description: The current implementation of _setPauseStatus does not update the isP2PDisabled. When _-isPaused = false this is not a real problem because once all the flags are enabled (everything is paused), all the operations will be blocked at the root of the execution of the process.

There might be cases instead where isP2PDisabled and the other flags were disabled for a market and Morpho want to enable all of them, resuming all the operations and allowing the users to continue P2P usage. In this case, Morpho would only resume operations without allowing the users to use the P2P flow.

```
function _setPauseStatus(address _poolToken, bool _isPaused) internal {
    Types.MarketPauseStatus storage pause = marketPauseStatus[_poolToken];

    pause.isSupplyPaused = _isPaused;
    pause.isBorrowPaused = _isPaused;
    pause.isWithdrawPaused = _isPaused;
    pause.isRepayPaused = _isPaused;
    pause.isLiquidateCollateralPaused = _isPaused;
    pause.isLiquidateBorrowPaused = _isPaused;

    // ... event emissions
}
```

Recommendation: Consider also updating the isP2PDisabled flag if the intended behavior of the function is to update all the flags of a market.

Morpho: We acknowledge the risk and won't be fixing this issue.

Spearbit: Acknowledged.

5.5.2 Differences between Morpho and Aave liquidate validation logic

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L204-L287, aave-v2/ExitPositionsManager.sol#L643, aave-v2/ExitPositionsManager.sol#L468

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between the logic

Note: Morpho re-implements the liquidate function as a mix of

- repay + supply operations on Aave executed inside _unsafeRepayLogic where needed
- withdraw + borrow operations on Aave executed inside _unsafeWithdrawLogic where needed

From _unsafeRepayLogic (repay + supply on pool where needed)

- Because _unsafeRepayLogic internally call aave.supply the whole tx could fail in case the supplying has been disabled on Aave (isFrozen == true) for the _poolTokenBorrowed
- Morpho is not checking that the Aave borrowAsset has isActive == true
- Morpho do not check that remainingToRepay.rayDiv(poolIndexes[_poolToken].poolBorrowIndex) > 0. Trying to repay that amount to Aave would make the whole tx revert

- Morpho do not check that `remainingToSupply.rayDiv(poolIndexes[_poolToken].poolSupplyIndex) > 0`. Trying to borrow that amount to Aave would make the whole tx revert

From `_unsafeWithdrawLogic` (`withdraw` + `borrow` on pool where needed)

- Because `_unsafeWithdrawLogic` internally calls `aave.borrow` the whole tx could fail in case the borrowing has been disabled on Aave (`isFrozen == true` or `borrowingEnabled == false`) for the `_poolTokenCollateral`
- Morpho is not checking that the Aave `collateralAsset` has `isActive == true`
- Morpho do not check that `remainingToWithdraw.rayDiv(poolIndexes[_poolToken].poolSupplyIndex) > 0`. Trying to withdraw that amount from Aave would make the whole tx revert
- Morpho do not check that `remainingToBorrow.rayDiv(poolIndexes[_poolToken].poolBorrowIndex) > 0`. Trying to borrow that amount from Aave would make the whole tx revert

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.3 Differences between Morpho and Aave `repay` validation logic

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L181-L197, aave-v2/ExitPositionsManager.sol#L643

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between the logic

Note: Morpho re-implement the `repay` function as a mix of `repay` + `supply` operations on Aave where needed

- Both Aave and Morpho are not handling `ERC20` token with fees on transfer
- Because `_unsafeRepayLogic` internally call `aave.supply` the whole tx could fail in case the supplying has been disabled on Aave (`isFrozen == true`)
- Morpho is not checking that the Aave market has `isActive == true`
- Morpho do not check that `remainingToRepay.rayDiv(poolIndexes[_poolToken].poolBorrowIndex) > 0`. Trying to repay that amount to Aave would make the whole tx revert
- Morpho do not check that `remainingToSupply.rayDiv(poolIndexes[_poolToken].poolSupplyIndex) > 0`. Trying to supply that amount to Aave would make the whole tx revert

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.4 Differences between Morpho and Aave `withdraw` validation logic

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L154-L173, aave-v2/ExitPositionsManager.sol#L468

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between the logic

Note: Morpho re-implement the `withdraw` function as a mix of `withdraw + borrow` operations on Aave where needed

- Both Aave and Morpho are not handling `ERC20` token with fees on transfer
- Because `_unsafeWithdrawLogic` internally calls `aave.borrow` the whole tx could fail in case the borrowing has been disabled on Aave (`isFrozen == true` or `borrowingEnabled == false`)
- Morpho is not checking that the Aave market has `isActive == true`
- Morpho do not check that `remainingToWithdraw.rayDiv(poolIndexes[_poolToken].poolSupplyIndex) > 0`. Trying to withdraw that amount from Aave would make the whole tx revert
- Morpho do not check that `remainingToBorrow.rayDiv(poolIndexes[_poolToken].poolBorrowIndex) > 0`. Trying to borrow that amount from Aave would make the whole tx revert

Note 1: Aave is NOT checking that the market `isFrozen`. This means that users can withdraw even if the market is active but frozen

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.5 Differences between Morpho and Aave `borrow` validation logic

Severity: Informational

Context: aave-v2/EntryPositionsManager.sol#L188-L280

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between the logics

Note: Morpho re-implement the `borrow` function as a mix of `withdraw` + `borrow` operations on Aave where needed

- Both Aave and Morpho are not handling `ERC20` token with fees on transfer
- Morpho is not checking that the Aave market has `isFrozen == false` (check done by Aave on the borrow operation), users could be able to borrow in P2P even if the borrow is paused on Aave (`isFrozen == true`) because Morpho would only call the `aave.withdraw` (where the frozen flag is not checked)
- Morpho do not check if market is active (would `borrowingEnabled == false` if market is not active?)
- Morpho do not check if market is frozen (would `borrowingEnabled == false` if market is not frozen?)
- Morpho do not check that `healthFactor > GenericLogic.HEALTH_FACTOR_LIQUIDATION_THRESHOLD`
- Morpho do not check that `remainingToBorrow.rayDiv(poolIndexes[_poolToken].poolBorrowIndex) > 0`. Trying to borrow that amount from Aave would make the whole tx revert

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.6 Differences between Morpho and Aave `supply` validation logic

Severity: Informational

Context: aave-v2/EntryPositionsManager.sol#L90-L182

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between the logics

Note: Morpho re-implement the `supply` function as a mix of `repay` + `supply` operations on Aave where needed

- Both Aave and Morpho are not handling ERC20 token with fees on transfer
- Morpho is not checking that the Aave market has `isFrozen == false`, users could be able to supply in P2P even if the supply is paused on Aave (`isFrozen == true`) because Morpho would only call the `aave.repay` (where the frozen flag is not checked)
- Morpho is not checking if `remainingToSupply.rayDiv( poolIndexes[_poolToken].poolSupplyIndex ) === 0`. Trying to supply that amount to Aave would make the whole tx revert

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags:

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.7 Morpho should avoid creating a new market when the underlying Aave market is frozen

Severity: Informational

Context: aave-v2/MorphoGovernance.sol#L473

Description: In the current implementation of Aave `MorphoGovernance.createMarket` the function is only checking if the `AToken` is in active state.

Morpho should also check if the `AToken` is **not** in a frozen state. When a market is frozen, many operations on the Aave side will be prevented (reverting the transaction).

Recommendation: Consider adding a check on the `getFrozen()` flag when creating a new market

Morpho: The recommendation will only be added to the off-chain checklist followed before creating a new market PR 6.

Spearbit: Acknowledged.

5.5.8 Differences between Morpho and Compound `liquidate` validation logic

Severity: Informational

Context: PositionsManager.sol#L487-L511

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

Note: Morpho liquidation does not directly call `compound.liquidate` but acts as a `repay` + `withdraw` operation.

By reviewing both logic, we have noticed that there are some differences between the logic

- Morpho does not check Compound `seizeGuardianPaused` because it is not implementing a "real" liquidate on compound, but it's emulating it as a "repay" + "withdraw".
  - Morpho should anyway monitor off-chain when the value of `seizeGuardianPaused` changes to true. Which are the scenarios for which Compound decides to block liquidations (across all cTokens)? When this happens, is Compound also pausing all the other operations?
  - **[Open question]** Should Morpho pause liquidations when the `seizeGuardianPaused` is true?
- Morpho is not reverting if `msg.sender === borrower`
- Morpho does not check if `_amount > 0`
- Compound revert if `amountToSeize > userCollateralBalance`, Morpho does not revert and instead uses `min(amountToSeize, userCollateralBalance)`

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Aave"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags:

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.9 `repayLogic` in Compound `PositionsManager`should revert if `toRepay` is equal to zero

Severity: Informational

Context: PositionsManager.sol#L471

Description: The current implementation of `repayLogic` is correctly reverting if `_amount == 0` but is not reverting if `toRepay == 0`. The value inside `toRepay` is given by the `min` value between `_getUserBorrowBalanceInOf(_poolToken, _onBehalf)` and `_amount`.

If the `_onBehalf` user has zero debt, `toRepay` will be initialized with zero.

Recommendation: Consider reverting if `toRepay == 0`

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.5.10 Differences between Morpho and Compound `supply` validation logic

Severity: Informational

Context: compound/PositionsManager.sol#L240-L243

Description: The Morpho approach is to mimic 1:1 the logic of the underlying protocol, including all the logic and sanity checks that are done before executing a user's action. On top of the protocol's logic, Morpho has its own logic.

By reviewing both logic, we have noticed that there are some differences between them;

- Compound is handling ERC20 tokens that could have transfer fees, Morpho is not doing it right now, see*"ERC20 with transfer's fee are not handled by `PositionManager`"*.
- Morpho is not checking if the underlying Compound market has been paused for the supply action (see `mintGuardianPaused[token]`). This means that even if the Compound supply is paused, Morpho could allow users to supply in the P2P.
- Morpho is not checking if the market on both Morpho and Compound has been deprecated. If the deprecation flag is intended to be true for a market that will be removed in the next future, probably Morpho should not allow users to provide collateral for such a market.

More information about detailed information can be found in the discussion topic *"Differences in actions checks between Morpho and Compound"*.

Recommendation: Consider implementing the missing logic/sanity checks or documenting why those checks should not be added to Morpho's implementation.

Morpho: Here are the motivations behind not implementing the flags

1. No amount of checks will be sufficient, and the Morpho Operator should be proactive about pausing before the underlying pool does.
2. Hardcoding can actually reduce flexibility in Morpho's pausing discretion since one action on Morpho can actually constitute multiple types of interactions with the pool.
3. Some checks are gas intensive as information from the pool needs to be fetched.

Spearbit: Acknowledged.

5.5.11 Consider creating a documentation that covers all the Morpho own flags, lending protocol's flags and how they interact/override each other

Severity: Informational

Context: General

Description: Both Morpho and Aave/Compound have their own flags to check before allowing a user to interact with the protocols. Usually, Morpho has decided to follow the logic to map 1:1 the implementation of the underlying protocol validation.

There are some examples also where Morpho has decided to override some of their own internal flags

For example, in the Aave aave-v2/ExitPositionsManager.liquidateLogic even if a Morpho market has been flagged as "deprecated" (user can be liquidated without being insolvent) the liquidator would not be able to liquidate the user if the liquidation logic has been paused.

Recommendation: Morpho should create in-depth documentation that explains all these flags and how they interact with each other and which was the scenario for which Morpho has decided not to follow the protocol's behavior (if this scenario exists).

The documentation will be very useful to track where and when those flags interact with each other and what could be the possible outcome of a change decision.

Morpho: Done internally.

Spearbit: Fixed.

5.5.12 Missing natspec or typos in natspec

Severity: Informational

Context: General

Description: - Updated the natspec `updateP2PIndexes` replacing "exchangeRatesStored()" with "exchangeRate-Stored()"

- Updated the natspec `_updateP2PIndexes` replacing "exchangeRatesStored()" with "exchangeRateStored()"
- Updated the natspec for `event MarketCreated` replacing "_poolToken" with "_p2pIndexCursor"

Recommendation: Implement the recommended natspec fixes suggested in the description.

Morpho: The recommendations have been implemented in the PR 1553

Spearbit: Fixed.

5.5.13 Removed unused "named" return parameters from functions

Severity: Informational

Context: - MorphoUtils.sol#L42-L48

- MorphoUtils.sol#L50-L54

Description: Some functions in the codebase are defining "named" functions parameter that are not used explicitly inside the code. This could lead to future changes to return wrong values if the "explicit return" statement is removed and the function returns the "default" values (based on the variable type) of the "named" parameter.

Recommendation: Remove the "named" parameter and only use the explicit return statement.

Morpho: The issues referenced in the "context" sessions have been fixed with PR 1548.

Spearbit: Acknowledged. Those references were just an example of other cases that have been found in the overall codebase. Morpho should consider checking the whole codebase and refactoring the code, removing the "named return parameters" where possible or needed.

5.5.14 Consider merging the code of `CompoundMath` libraries and use only one

Severity: Informational

Context: - compound/InterestRatesManager.sol#L7

- compound/MorphoUtils.sol#L6
- RewardsManager.sol#L7

Description: The current codebase uses `libraries/CompoundMath.sol` but there's already an existing solidity library with the same name inside the package `@morpho-dao/morpho-utils`

For better code clarity, consider merging those two libraries and only importing the one from the external package. Be aware that the current implementation inside the `@morpho-dao/morpho-utils CompoundMath mul` and `div` function uses low-level yul and should be tested, while the library used right now in the code use "high level" solidity.

Recommendation: Consider merging the code between `libraries/CompoundMath.sol` and `@morpho-dao/morpho-utils CompoundMath` and use only the one imported from the external package.

Morpho: `CompoundMath` library has been removed from the codebase and replaced by `morpho-utils` external libraries (mix of `Math` and `CompundMath`). Note that while the previous implementation of the original implementation of `CompoundMath` was using plain solidity, the new one (and the replaced Math functions) are using low-level yul.

See PR 1521 and PR 1615.

Spearbit: Verified.

5.5.15 Consider reverting the creation of a deprecated market in Compound

Severity: Informational

Context: compound/MorphoGovernance.sol#L442

Description: Compound has a mechanism that allows the Governance to set a specific market as "deprecated". Once a market is deprecated, all the borrows can be liquidated without checking whether the user is solvent or not. Compound currently allows users to enter (to supply and borrow) a market.

In the current version of `MorphoGovernance.createMarket`, Morpho governance is not checking whether a market is already deprecated on compound before entering it and creating a new Morpho-market. This would allow a Morpho user to possibly supply or borrow on a market that has been already deprecated by compound.

Recommendation: Consider reverting the creation of a Morpho market if the cToken has been deprecated on Compound.

Morpho: It's unlikely that the governance would list a deprecated market and it does prevent the underlying pool to deprecate a market later either, we acknowledge this issue.

Spearbit: Acknowledged.

5.5.16 Document `HeapOrdering`

Severity: Informational

Context: HeapOrdering.sol#L66

Description: Morpho uses a non-standard Heap implementation for their Aave P2P matching engine. The implementation only correctly sorts `_maxSortedUsers / 2` instead of the expected `_maxSortedUsers`. Once the `_maxSortedUsers` is reached, it halves the size of the heap, cutting the last level of leaves of the heap. This is done because a naive implementation that would insert new values at `_maxSortedUsers` (once the heap is full) and shift them up, then decrease the `size` to `_maxSortedUsers - 1` again, would end up concentrating all new values on the same single path from the leaf to the root node. Cutting off the last level of nodes of the heap is a heuristic to remove low-value nodes (because of the heap property) while at the same time letting new values be shifted up from different leaf locations. In the end, the goal this tries to achieve is that more high-value nodes are stored in the heap and can be used for the matching engine.

Recommendation: Document your heap implementation and what it tries to achieve. The `_maxSortedUsers` value should be of the form `2^k - 1` to have a full binary tree optimizing the distribution in the last step. The current version that morpho-v1 uses as its dependency does not have as many tests as the latest version of this data structure. Consider updating to the newer, more-tested version. Consider fuzz-testing this data structure to ensure it works with random values in unpredictable insertion order.

Morpho: An issue has been created for this: issue 109 but not implemented yet.

Spearbit: Acknowledged.

5.5.17 Consider removing the Aave-v2 reward management logic if it is not used anymore

Severity: Informational

Context:

- aave-v2/Morpho.sol#L148-L172
- aave-v2/Morpho.sol#L17-L26
- aave-v2/MorphoGovernance.sol
- aave-v2/MatchingEngine.sol#L314-L321
- aave-v2/MatchingEngine.sol#L340-L351

Description: If the current aave-v2 reward program has ended and the Aave protocol is not re-introducing it anytime soon (if not at all) consider removing the code that currently is handling all the logic behind claiming rewards from the Aave lending pool for the supplied/borrow assets.

Removing that code would make the codebase cleaner, reduce the attack surface and possibly revert in case some of the state variables are incorrectly miss configured (rewards management on Morpho is activated but Aave is not distributing rewards anymore).

Recommendation: Consider removing the Aave-v2 reward management logic if it is not used anymore.

Morpho: The recommendations have been implemented in PR 1545. The storage variables have been marked with a "Deprecated" comment, and the `claimRewards` function in the Morpho contract now has an empty body.

Spearbit: Fixed. As an additional recommendation, Morpho should notify, with large advance, all the integrators that rely on these APIs about the changes that will be deployed.

5.5.18 Avoid shadowing state variables

Severity: Informational

Context: aave-v2/InterestRatesManager.sol#L61 aave-v2/EntryPositionsManager.sol#L99 aave-v2/EntryPositionsManager.sol#L194 aave-v2/MorphoGovernance.sol#L444 aave-v2/MorphoGovernance.sol#L482

Description: Shadowing state or global variables could lead to potential bugs if the developer does not treat them carefully. To avoid any possible problem, every local variable should avoid shadowing a state or global variable name.

Recommendation: Rename the local scope variable with a different name than the storage variable to avoid shadowing.

Morpho: We won't change variable naming as we consider it not worth doing it right now. Thus, we acknowledge this issue.

Spearbit: Acknowledged.

5.5.19 Governance setter functions do not check current state before updates

Severity: Informational

Context: aave-v2/MorphoGovernance.sol#L178-L413 compound/MorphoGovernance.sol

Description: In `MorphoGovernance.sol`, many of the setter functions allow the state to be changed even if it is already set to the passed-in argument. For example, when calling `setP2PDisabled`, there are no checks to see if the `_poolToken` is already disabled, or does not allow unnecessary state changes.

Recommendation: The current state should be checked on important setter functions and if the new state to change is a duplicate, then the function should revert.

Morpho: We do not think it's worth adding this check as the end result is the same whether there is a revert or not, so we acknowledge this issue.

Spearbit: Acknowledged.

5.5.20 Emit event for amount of dust used to cover withdrawals

Severity: Informational

Context: aave-v2/PositionsManagerUtils.sol#L62

Description: Consider emitting an event that includes the amount of dust that was covered by the contract balance. A couple of ways this could be used:

- Trigger an alert whenever it exceeds a certain threshold so you can inspect it, and pause if a bug is found or a threshold is exceeded.
- Use this value as part of your overall balance accounting to verify everything adds up.

Recommendation: Emit an event when a withdrawal is done.

Morpho: The amount is assumed to be very small and because of this, we considered it is not worth adding an event for this purpose.

Spearbit: Acknowledged.

5.5.21 Break up long functions into smaller composable functions

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L204 aave-v2/ExitPositionsManager.sol#L336 aave-v2/ExitPositionsManager.sol#L491 aave-v2/EntryPositionsManager.sol#L90 aave-v2/EntryPositionsManager.sol#L188

Description: A few functions are 100+ lines of code which makes it more challenging to initially grasp what the function is doing. You should consider breaking these up into smaller functions which would make it easier to grasp the logic of the function, while also enabling you to easily unit test the smaller functions.

Recommendation: Refactor these long functions to instead be comprised of smaller functions. If all is done correctly, the entire existing test suite should pass. Add new tests for the new functions.

Morpho: Because this requires large changes to the codebase we prefer to acknowledge this recommendation.

Spearbit: Acknowledged.

5.5.22 Remove unused struct members

Severity: Informational

Context:aave-v2/ExitPositionsManager.sol#L140

Description: The `HealthFactorVars` struct contains three attributes, but only the `userMarkets` attribute is ever set or used. These should be removed to increase code readability.

Recommendation: Verify the other two attributes `i` and `numberOfMarketsCreated` are not needed, and remove them if not needed.

Morpho: Fixed in PR 1557.

Spearbit: Verified.

5.5.23 Remove unused struct

Severity: Informational

Context: aave-v2/EntryPositionsManager.sol#L76

Description: There is an unused struct `BorrowAllowedVars`. This should be removed to improve code readability.

Recommendation: Verify it is not needed, and remove if not needed.

Morpho: Fixed in PR 1557.

Spearbit: Verified.

5.5.24 No validation check on prices fetched from the oracle

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L259-L260 aave-v2/MorphoUtils.sol#L272

Description: Currently in the `liquidateLogic` function when fetching the `borrowedTokenPrice` and `collateral-Price` from the `oracle`, the return value is not validated. This is due to the fact that the underlying protocol does not do this check either, but the fact that the underlying protocol does not do validation should not deter Morpho from performing validation checks on prices fetched from oracles.

Also, this check is done in the Compound `PositionsManager.sol` here so for code consistency, it should also be done in Aave-v2.

Recommendation: The `borrowedTokenPrice` and `collateralPrice` fetched from the oracle should be validated and reverted if they are zero.

Morpho: We'll continue to follow the behavior of Aave and thus acknowledge this issue.

Spearbit: Acknowledge.

5.5.25 `onBehalf` argument can be set as the Morpho protocols address

Severity: Informational

Context: aave-v2/EntryPositionsManager.sol#L93 compound/PositionsManager.sol#L236

Description: When calling the `supplyLogic` function, currently the `_onBehalf` argument allows a user to supply funds on behalf of the Morpho protocol itself. While this does not seem exploitable, it can still be a cause for user error and should not be allowed.

Recommendation: The `supplyLogic` function should revert if the `_onBehalf` argument address is the Morpho protocol itself.

Morpho: We did not find any potential issue for this case, thus we acknowledge the issue.

Spearbit: Acknowledged.

5.5.26 `maxSortedUsers` has no upper bounds validation and is not the same in Compound/Aave-2

Severity: Informational

Context: compound/MorphoGovernance.sol#L170 aave-v2/MorphoGovernance.sol

Description: In `MorphoGovernance.sol`, the `maxSortedUsers` function has no upper bounds limit put in place. The `maxSortedUsers` is the number of users to sort in the data structure. Also, while this function has the `MaxSortedUsersCannotBeZero()` check in Aave-v2, the Compound version is missing this same error check.

Recommendation: Consider setting an upper bounds limit on the `maxSortedUsers` number so as not to run into gas issues when sorting user data in the data structure. Also, the `MaxSortedUsersCannotBeZero()` check should be added in the Compound version of this function as well for code consistency.

Morpho: We acknowledge the issue but will not implement the recommendation.

Spearbit: Acknowledged.

5.5.27 Consider adding the compound revert error code inside Morpho custom error to better track the revert reason

Severity: Informational

Context: MorphoGovernance.sol#L443 PositionsManager.sol#L927 PositionsManager.sol#L937 PositionsManager.sol#L945 PositionsManager.sol#L970

Description: On Compound, when an error condition occurs, usually (except in extreme cases) the transaction is not reverted, and instead an error code (code !== 0) is returned.

Morpho correctly reverts with a custom error when this happens, but is not reporting the error code returned by Compound. By tracking, as an event parameter, the error code, Morpho could better monitor when and why interactions with Compound are failing.

Recommendation: Consider adding the Compound returned error to Morpho's custom error.

Morpho: Fixed in PR 1550.

Spearbit: Fixed.

5.5.28 `liquidationThreshold` variable name can be misleading

Severity: Informational

Context: aave-v2/ExitPositionsManager.sol#L678

Description: The `liquidationThreshold` name in Aave is a percentage. The `values.liquidationThreshold` variable used in Morpho's `_getUserHealthFactor` is in "value units" like debt:

```
values.liquidationThreshold = assetCollateralValue.percentMul(assetData.liquidationThreshold);.
```

Recommendation: Consider renaming the variable to avoid confusion. For example, `liquidationThreshold-Value`.

Morpho: Fixed in PR 1547 and PR 1559.

Spearbit: Verified.

5.5.29 Users can be liquidated on Morpho at any time when the deprecation flag is set by governance

Severity: Informational

Context: aave-v2/MorphoGovernance.sol#L395, compound/MorphoGovernance.sol#L372, aave-v2/ExitPositionsManager.sol#L706

Description: Governance can set a `deprecation` flag on Compound and Aave markets, and users on this market can be liquidated by anyone even if they're sufficiently over-collateralized. Note that this deprecation flag is independent of Compound's own deprecation flags and can be applied to any market.

Recommendation: Users should be aware of this. Clearly communicate when you deprecate a market and give enough time for users to unwind their positions on the markets to be deprecated.

Morpho: No changes have been made in the code but users will be warn in any case since it requires governance approval to do so.

Spearbit: Acknowledged.

5.5.30 Refactor `_computeP2PIndexes` to use `InterestRateModel`'s functions

Severity: Informational

Context: aave-v2/InterestRatesManager.sol#L113, compound/InterestRatesManager.sol#L100, aave-v2/InterestRatesModel.sol#L49

Description: The `InterestRatesManager` contracts' `_computeP2PIndexes` functions currently reimplement the interest rate model from the `InterestRatesModel` functions.

Recommendation: Consider refactoring the `InterestRatesManager._computeP2PIndexes` to use `Intere-stRatesModel` functions like `computeGrowthFactors`, `computeP2PSupplyIndex`, and `computeP2PBorrowIndex`. This would also guarantee that the `lens` contracts indeed use the same model that the contract uses as they use the mentioned `InterestRatesModel` functions.

Morpho: Fixed in PR 1537.

Spearbit: Verified.