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

May 4, 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-aave-v3 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-aave-v3 protocol. In this period of time a total of **36** issues were found.

An additional two-day engagement took place on April 20th and April 21st, during which several issues were reviewed.

Summary

Project Name	Morpho
Repository	morpho-aave-v3
Commit	76525d...0855
Type of Project	Lending and Borrowing, DeFi
Audit Timeline	Feb 13 - Feb 24
Two week fix period	Feb 24 - March 10
April review	April 20 - April 21
April review commit	51afdd...9dca

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	4	4	0
High Risk	5	3	2
Medium Risk	7	5	2
Low Risk	4	3	1
Gas Optimizations	3	3	0
Informational	13	7	6
Total	**36**	**25**	**11**

5 Findings

5.1 Critical Risk

5.1.1 Side effects of LTV = 0 assets: Morpho's users will not be able to withdraw (collateral and "pure" supply), borrow and liquidate

Severity: Critical Risk

Context: PositionsManager.sol#L120-L121, PositionsManager.sol#L180, PositionsManager.sol#L213, PositionsManager.sol#L261

Description: When an AToken has `LTV = 0`, Aave restricts the usage of some operations. In particular, if the user owns at least one AToken as collateral that has `LTV = 0`, operations could revert.

1) Withdraw: if the asset withdrawn is collateral, the user is borrowing something, the operation will revert if the withdrawn collateral is an AToken with `LTV > 0`.
2) Transfer: if the `from` is using the asset as collateral, is borrowing something and the asset transferred is an AToken with `LTV > 0` the operation will revert.
3) Set the reserve of an AToken as not collateral: if the AToken you are trying to set as non-collateral is an AToken with `LTV > 0` the operation will revert.

Note that all those checks are done on top of the "normal" checks that would usually prevent an operation, depending on the operation itself (like, for example, an HF check).

While a "normal" Aave user could simply withdraw, transfer or set that asset as non-collateral, Morpho, with the current implementation, cannot do it. Because of the impossibility to remove from the Morpho wallet the "poisoned AToken", part of the Morpho mechanics will break.

- Morpho's users could not be able to withdraw both collateral and "pure" supply
- Morpho's users could not be able to borrow
- Morpho's users could not be able to liquidate
- Morpho's users could not be able to claim rewards via `claimRewards` if one of those rewards is an `AToken` with `LTV > 0`

Recommendation: Morpho should avoid listing as markets `ATokens` with `LTV = 0` or `ATokens` that soon will be `LTV = 0`.

In case Morpho has already created markets for tokens that will be configured with `LTV = 0` a well-detailed and tested procedure to reach a state that prevents the listed side effects should be applied as soon as possible.

The final goal of Morpho should be to arrive at a point where those markets are

- Paused.
- Have no supplied/borrow balance owned by the users.
- Have only 1 wei of collateral balance owned by Morpho.
- Have the reserve set as non-collateral while remaining overall healthy.

The last two points are to avoid possible griefing attacks.

Morpho: Fixed in PR 569.

Spearbit: The fix implements a mechanism that allows Morpho to set an asset as collateral/not collateral on Morpho and Aave. This implementation is needed to handle the edge case where an asset LTV is set to zero by the Aave Governance.

Without setting an LTV = 0 asset as non-collateral on Aave, some core mechanics of Morpho's protocol would break. While this PR solve this specific issue, all the side effects described in the issue still remain true.

When an asset is set to `isCollateral = false` on Morpho or has `LTV = 0` on Aave, Morpho's user's LTV and HF will be reduced because Morpho is treating that asset not as collateral anymore. The behavior has the following consequences for Morpho's users:

- User could not be able to borrow anymore (because of reduced LTV).
- User could not be able to withdraw anymore (because of reduced HF).
- User could be liquidable (because of reduced HF).
- Increase the possibility, in case of liquidation, to liquidate the whole debtor's collateral (because of reduced HF).
- While the asset is not threaded as collateral anymore, it can still be sized during the liquidation process.

Note that in case LTV = 0, the same user on Aave would have a different situation because on Aave, in this specific scenario, only the LTV is reduced and not the HF.

The PR is lacking documentation of this behavior and the differences between Morpho and Aave in this scenario. The PR is also lacking a well-documented procedure that the users should follow both before and after the LTV = 0 edge case to avoid being liquidated or incur in any of those side effects once Morpho's has set the asset as not-collateral or Aave has set the LTV to zero.

Because the PR does not solve the user's side effects, Morpho should consider documenting them and provide a well-documented procedure that the users should follow for the issue's scenario. Morpho should also consider implementing some UI/UX mechanism that properly alerts users to take those actions for assets that will soon be set to `isCollateral = false` on Morpho or `LTV = 0` on Aaave.

5.1.2 Morpho is vulnerable to attackers sending `LTV = 0` collateral tokens, supply/supplyCollateral, borrow and liquidate operations could stop working

Severity: Critical Risk

Context:

- Morpho related context: PositionsManager.sol#L120-L121, PositionsManager.sol#L180, PositionsManager.sol#L213, PositionsManager.sol#L261
- Aave related context: ValidationLogic.sol#L605-L608, SupplyLogic.sol#L146-L156, SupplyLogic.sol#L194-L204, SupplyLogic.sol#L280-L290

Description: When an AToken has `LTV = 0`, Aave restricts the usage of some operations. In particular, if the user owns at least one AToken as collateral that has `LTV = 0`, these operations could revert

1) Withdraw: if the asset withdrawn is collateral, the user is borrowing something, the operation will revert if the withdrawn collateral is an AToken with `LTV > 0`
2) Transfer: if the `from` is using the asset as collateral, is borrowing something and the asset transferred is an AToken with `LTV > 0` the operation will revert
3) Set the reserve of an AToken as not collateral: if the AToken you are trying to set as non-collateral is an AToken with `LTV > 0` the operation will revert

Note that all those checks are done on top of the "normal" checks that would usually prevent an operation, depending on the operation itself (like, for example, an HF check).

In the attack scenario, the bad actor could simply supply an underlying that is associated with an `LTV = 0` AToken and transfer it to the Morpho contract. If the victim does not own any balance of the asset, it will be set as collateral and the victim will suffer from all the side effects previously explained.

While a "normal" Aave user could simply withdraw, transfer or set that asset as non-collateral, Morpho, with the current implementation, cannot do it. Because of the impossibility to remove from the Morpho wallet the "poisoned AToken", part of the Morpho mechanics will break.

- Morpho's users could not be able to withdraw both collateral and "pure" supply.

- Morpho's users could not be able to borrow.
- Morpho's users could not be able to liquidate.
- Morpho's users could not be able to claim rewards via `claimRewards` if one of those rewards is an `AToken` with `LTV > 0`.

Recommendation: There are multiple possible solutions that could be explored by Morpho. Each of the solutions has pros, cons, and possible side effects that should be carefully evaluated.

One possible solution that Morpho could explore after elaborating all the pros/cons/side effects and possible problems could be to allow the DAO to set the "poison asset" as non-collateral. If the asset is part of Morpho's markets, they must be sure that Morpho's Aave position remains healthy. If the asset is not part of Morpho's markets, there should be no problem regarding the health factor.

A second possible solution could be to allow the DAO to withdraw/transfer the "poisoned token" if the token does not belong to an existing market (this is to prevent touching the user's assets). If the token belongs to Morpho's market, the solution is trickier and should probably fall back to the first solution without touching the user's balance in an active way.

Because of how Aave behaves, it should be noted that if Morpho withdraws/transfer all the balance of the LTV = 0 asset, the attacker can repeat the griefing attack by transferring again the "poisoned asset" to Morpho that would be re-enabled as collateral by default. Because of this, probably the second solution should be avoided.

It should be noted that the problems described above could exist for assets that already belong to the existing Morpho market. This scenario will be handled in a separate issue.

Morpho: Fixed by P5 569 followed by P5 768.

In case of emergency we'll follow the guidelines written in this notion link.

At deployment we sent aTokens of all markets to the Morpho contract and disabled them as collateral on pool so that this attack cannot be performed on the current listed assets on Aave.

Also, Aave is planning an upgrade that would fix this issue.

Spearbit: Fixed.

5.1.3 Morpho is not correctly handling the asset price in `_getAssetPrice` when `isInEMode == true` but `priceSource` is `addres(0)`

Severity: Critical Risk

Context: MorphoInternal.sol#L524-L536

Description: The current implementation of `_getAssetPrice` returns the asset's price based on the value of `isInEMode`

```
function _getAssetPrice(address underlying, IAaveOracle oracle, bool isInEMode, address priceSource)
    internal
    view
    returns (uint256)
{
    if (isInEMode) {
        uint256 eModePrice = oracle.getAssetPrice(priceSource);

        if (eModePrice != 0) return eModePrice;
    }

    return oracle.getAssetPrice(underlying);
}
```

As you can see from the code, if `isInEMode` is equal to `true` they call `oracle.getAssetPrice` no matter what the value of `priceSource` that could be `address(0)`.

If we look inside the `AaveOracle` implementation, we could assume that in the case where `asset` is `address(0)` (in this case, Morpho pass `priceSource _getAssetPrice` parameter) it would probably return `_fallbackOracle.getAssetPrice(asset)`.

In any case, the `Morpho` logic diverges compared to what Aave implements. On Aave, if the user is not in e-mode, the e-mode oracle is `address(0)` or the asset's e-mode is not equal to the user's e-mode (in case the user is in e-mode), Aave always uses the asset price of the `underlying` and not the one in the e-mode `priceSource`.

The impact is that if no explicit eMode oracle has been set, Morpho might revert in price computations, breaking liquidations, collateral withdrawals, and borrows if the fallback oracle does not support the asset, or it will return the fallback oracle's price which is different from the price that Aave would use.

Recommendation: Morpho should use and return the e-mode price `eModePrice` only when `isInEMode` and `priceSource != address(0)`

Morpho: Recommendation implemented in PR 590.

Spearbit: Fixed.

5.1.4 Isolated assets are treated as collateral in Morpho

Severity: Critical Risk

Context: aave-v3/SupplyLogic.sol#L78, aave-v3/ValidationLogic.sol#L711, aave-v3/UserConfiguration.sol#L194, PositionsManagerInternal.sol#L408

Description: Aave-v3 introduced isolation assets and isolation mode for users:

> *"Borrowers supplying an isolated asset as collateral cannot supply other assets as collateral (though they can still supply to capture yield). Only stablecoins that have been permitted by Aave governance to be borrowable in isolation the mode can be borrowed by users utilizing isolated collateral up to a specified debt ceiling."*

The Morpho contract is intended *not* to be in isolation mode to avoid its restrictions. Supplying an isolated asset to Aave while there are already other (non-isolated) assets set as collateral will simply supply the asset to earn yield without setting it as collateral. However, Morpho will still set these isolated assets as collateral for the supplying user. Morpho users can borrow any asset against them which should not be possible:

- Isolated assets are by definition riskier when used as collateral and should only allow borrowing up to a specific debt ceiling.
- The borrows are not backed on Aave as the isolated asset is not treated as collateral there, lowering the Morpho Aave position's health factor and putting the system at risk of liquidation on Aave.

Recommendation: The current code assumes that a supplied asset is always set as collateral on Aave whenever a `supplyCollateral` action with its `_POOL.supplyToPool` call succeeds.

In addition to *Morpho can end up in isolation mode* which ensures that the system does not end up in isolation mode, reject isolated assets for `supplyCollateral` calls. Alternatively, check that the supplied asset is indeed set as collateral on Aave after the `_POOL.supplyToPool(underlying, amount)` call.

Morpho: Addressed with PR 569.

More information on edge cases and how we would handle can be found here.

Spearbit: Fixed.

5.2 High Risk

5.2.1 Morpho's logic to handle LTV = 0 AToken diverges from the Aave logic and could decrease the user's HF/borrowing power compared to what the same user would have on Aave

Severity: High Risk

Context: MorphoInternal.sol#L324, PositionsManager.sol#L118, PositionsManager.sol#L209-L211, PositionsManagerInternal.sol#L176-L192

Description: The current implementation of Morpho has a specific logic to handle the scenario where Aave sets the asset's LTV to zero. We can see how Morpho is handling it in the `_assetLiquidityData` function

```
function _assetLiquidityData(address underlying, Types.LiquidityVars memory vars)
    internal
    view
    returns (uint256 underlyingPrice, uint256 ltv, uint256 liquidationThreshold, uint256 tokenUnit)
{
    // other function code...

    // If the LTV is 0 on Aave V3, the asset cannot be used as collateral to borrow upon a breaking
↪   withdraw.
    // In response, Morpho disables the asset as collateral and sets its liquidation threshold
    // to 0 and the governance should warn users to repay their debt.
    if (config.getLtv() == 0) return (underlyingPrice, 0, 0, tokenUnit);

    // other function code...
}
```

The `_assetLiquidityData` function is used to calculate the number of assets a user can borrow and the maximum debt a user can reach before being liquidated. Those values are then used to calculate the user Health Factor.

The Health Factor is used to

- Calculate both if a user can be liquidated and in which percentage the collateral can be seized.
- Calculate if a user can withdraw part of his/her collateral.

The `debt` and `borrowable amount` are used in the Borrowing operations to know if a user is allowed to borrow the specified amount of tokens.

On Aave, this situation is handled differently. First, there's a specific distinction when the `liquidation threshold` is equal to zero and when the Loan to Value of the asset is equal to zero. Note that Aave enforces (on the configuration setter of a reserve) that `ltv` must be <= of `liquidationThreshold`, this implies that if the LT is zero, the LTV must be zero.

In the first case (liquidation threshold equal to zero) the collateral is not counted as collateral. This is the same behavior followed by Morpho, but the difference is that Morpho also follows it when the Liquidation Threshold is greater than zero.

In the second case (LT > 0, LTV = 0) Aave still counts the collateral as part of the user's total collateral but does not increase the user's borrowing power (it does not increase the average LTV of the user). This influences the user's health factor (and so all the operations based on it) but not as impactfully as Morpho is doing.

In conclusion, when the LTV of an asset is equal to zero, Morpho is not applying the same logic as Aave is doing, removing the collateral from the user's collateral and increasing the possibility (based on the user's health factor, user's debt, user's total collateral and all the asset's configurations on Aave) to

- Deny a user's collateral withdrawal (while an Aave user could have done it).
- Deny a user's borrow (while an Aave user could have done it).
- Make a user liquidable (while an Aave user could have been healthy).
- Increasing the possibility to allow the liquidator to seize the full collateral of the borrower (instead of 50%).

Recommendation: While the motivation of the Morpho approach is understandable, related to the side effects of an LTV = 0 asset in the Morpho context (see *"Side effects of LTV = 0 assets: Morpho's users will not be able to withdraw (collateral and "pure" supply), borrow and liquidate"* and *"Morpho is vulnerable to attackers sending LTV = 0 collateral tokens, supply/supplyCollateral, borrow and liquidate operations could stop working"*, Morpho should evaluate all the possible alternative solutions to avoid creating a worse user experience in this edge case scenario.

If there are no other possible options, Morpho should at least

- Add in-depth documentation in the code about this decision and all the side effects that it could have on the user positions
- Detail this behavior on their online documentation and warn the user

Morpho: Addressed in PR 569.

Spearbit: The PR implements a mechanism that allows the Morpho DAO to set an asset as collateral/not collateral on Morpho and Aave. This implementation is needed to handle the edge case where an asset LTV is set to zero by the Aave Governance.

Without setting an LTV = 0 asset as non-collateral on Aave, some core mechanics of Morpho's protocol would break. While this PR solves this specific issue, all the side effects described in the issue still remain true.

When an asset is set to `isCollateral = false` on Morpho or has `LTV = 0` on Aave, Morpho's user's LTV and HF will be reduced because Morpho is treating that asset not as collateral anymore. The behavior has the following consequences for Morpho's users:

- User could not be able to borrow anymore (because of reduced LTV).
- User could not be able to withdraw anymore (because of reduced HF).
- User could be liquidable (because of reduced HF).
- Increase the possibility, in case of liquidation, to liquidate the whole debtor's collateral (because of reduced HF).
- While the asset is not threaded as collateral anymore, it can still be sized during the liquidation process.

Note that in case LTV = 0, the same user on Aave would have a different situation because on Aave, in this specific scenario, only the LTV is reduced and not the HF.

The PR is lacking documentation of this behavior and the differences between Morpho and Aave in this scenario. The PR is also lacking a well documented procedure that the users should follow both before and after the LTV = 0 edge case to avoid being liquidated or incur in any of those side effects once Morpho's has set the asset as not-collateral or Aave has set the LTV to zero.

Because the PR does not solve the user's side effects, Morpho should consider documenting them and provide a well-documented procedure that the users should follow for the issue's scenario.

Morpho should also consider implementing some UI/UX mechanism that properly alerts users to take those actions for assets that will soon be set to `isCollateral = false` on Morpho or `LTV = 0` on Aaave.

Marked as Acknowledged.

5.2.2 `RewardsManager` does not take in account users that have supplied collateral directly to the pool

Severity: High Risk

Context: RewardsManager.sol#L436

Description: Inside `RewardsManager._getUserAssetBalances` Morpho is calculating the amount of the supplied and borrowed balance for a specific `user`. In the current implementation, Morpho is ignoring the amount that the user has supplied as collateral directly into the Aave pool. As a consequence, the user will be eligible for fewer rewards or even zero in the case where he/she has supplied only collateral.

Recommendation: When `asset == market.aToken`, `userAssetBalances[i].balance` should be equal to `_MORPHO.scaledPoolSupplyBalance(market.underlying, user) + _MORPHO.scaledCollateralBalance(market.underlying, user)`

Morpho: Recommendation implemented in PR 587.

Spearbit: Fixed.

5.2.3 Accounting issue when repaying P2P fees while having a borrow delta

Severity: High Risk

Context: PositionsManagerInternal.sol#L308, DeltasLib.sol#L88, MarketSideDeltaLib.sol#L63

Description: When repaying debt on Morpho, any potential borrow delta is matched first. Repaying the delta should involve both decreasing the `scaledDelta` as well as decreasing the `scaledP2PAmount` by the matched amount. [^1] However, the `scaledP2PAmount` update is delayed until the end of the repay function. The following `repayFee` call then reads the un-updated `market.deltas.borrow.scaledP2PAmount` storage variable leading to a larger estimation of the P2P fees that can be repaid.

The excess fee that is repaid will stay in the contract and not be accounted for, when it should have been used to promote borrowers, increase idle supply or demote suppliers. For example, there could now be P2P suppliers that should have been demoted but are not and in reality don't have any P2P counterparty, leaving the entire accounting system in a broken state.

- Example (all values are in underlying amounts for brevity.)

Imagine a borrow delta of `1000`, `borrow.scaledP2PTotal = 10,000` `supply.scaledP2PTotal = 8,000`, so the repayable fee should be `(10,000 - 1000) - (8,000 - 0) = 1,000`. Now a P2P borrower wants to repay 3000 debt:

1. Pool repay: no pool repay as they have no pool borrow balance.
2. Decrease p2p borrow delta: `decreaseDelta` is called which sets `market.deltas.borrow.scaledDelta = 0` (but does not update `market.deltas.borrow.scaledP2PAmount` yet!) and returns `matchedBorrowDelta = 1000`
3. `repayFee` is called and it computes `(10,000 - 0) - (8,000 - 1,000) = 2,000`. They repay more than the actual fee.

Recommendation: One way to resolve this issue is by having `MarketSideDeltaLib.decreaseDelta` decrease the `deltas.borrow.scaledP2PTotal` by the scaled repaid amount. The `market.deltas.decreaseP2P` call then only needs to reduce the P2P delta by `vars.toSupply + idleSupplyIncrease`. Consider adding tests for a scenario that has both a positive borrow delta and fees to repay.

Morpho: Fixed in PR 565.

Spearbit: Fixed, the `scaledP2PTotal` is now decreased before calling `repayFee`.

5.2.4 Repaying with ETH does not refund excess

Severity: High Risk

Context: WETHGateway.sol#L67

Description: Users can repay WETH Morpho positions with ETH using the `WETHGateway`. The specified repay `amount` will be wrapped to WETH before calling the Morpho function to repay the WETH debt. However, the Morpho repay function only pulls in `Math.min(_getUserBorrowBalanceFromIndexes(underlying, onBehalf, indexes), amount)`. If the user specified an `amount` larger than their debt balance, the excess will be stuck in the `WETHGateway` contract.

This might be especially confusing for users because the standard `Morpho.repay` function does not have this issue and they might be used to specifying a large, round value to be sure to repay all principal and accrued debt once the transaction is mined.

Recommendation: Compute the difference between the specified amount and the amount that was actually repaid, and refund it to the user.

```
uint256 excess = msg.value - _MORPHO.repay(_WETH, msg.value, onBehalf);
_unwrapAndTransferETH(excess, msg.sender);
```

Furthermore, consider adding `skim` functions that can send any stuck ERC20 or native balances to a recovery account, for example, the Morpho treasury (if defined).

Morpho: Fixed in PR 588 and PR 605.

Spearbit: Fixed.

5.2.5 Morpho can end up in isolation mode

Severity: High Risk

Context: aave-v3/SupplyLogic.sol#L78, aave-v3/ValidationLogic.sol#L711, aave-v3/UserConfiguration.sol#L194

Description: Aave-v3 introduced isolation assets and isolation mode for users:

> *"Borrowers supplying an isolated asset as collateral cannot supply other assets as collateral (though they can still supply to capture yield). Only stablecoins that have been permitted by Aave governance to be borrowable in isolation the mode can be borrowed by users utilizing isolated collateral up to a specified debt ceiling."*

The Morpho contract has a single Aave position for all its users and does therefore not want to end up in isolation mode due to its restrictions. The Morpho code would still treat the supplied non-isolation assets as collateral for their Morpho users, allowing them to borrow against them, but the Aave position does not treat them as collateral anymore. Furthermore, Morpho can only borrow stablecoins up to a certain debt ceiling.

Morpho can be brought into isolation mode:

- Up to deployment, an attacker maliciously sends an isolated asset to the address of the proxy. Aave sets assets as collateral when transferred, such that the Morpho contract already starts out in isolation mode. This can even happen before deployment by precomputing addresses or simply frontrunning the deployment. **This attack also works if Morpho does not intend to create a market for the isolated asset**.
- Upon deployment and market creation: An attacker or unknowing user is the first to supply an asset and this asset is an isolated asset, Morpho's Aave position automatically enters isolation mode.
- At any time if an isolated asset is the only collateral asset. This can happen when collateral assets are turned off on Aave, for example, by withdrawing (or liquidating) the entire balance.

Recommendation: Never end up in isolation mode. Upon deployment, consider setting a non-isolation asset as collateral by using `pool.supply` on behalf of the contract, or simply by sending an `aToken` of a non-isolated asset to the contract address. Also, consider adding a function calling `setReserveAsCollateral` to be able to turn off collateral *for isolated assets* at any time.

Note: this function reverts if there is no balance.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.3 Medium Risk

5.3.1 Collateral setters for Morpho / Aave can end up in a deadlock

Severity: Medium Risk

Context: MorphoSetters.sol#L87-L107

Description: One can end up in a deadlock where changing the Aave pool or Morpho collateral state is not possible anymore because it can happen that Aave automatically turns the collateral asset off (for example, when withdrawing everything / getting liquidated).

Imagine a collateral asset is turned `on` for both protocols:

```
setAssetIsCollateralOnPool(true)
setAssetIsCollateral(true)
```

Then, a user withdraws everything on Morpho / Aave, and Aave automatically turns it `off`. It's `off` on Aave but `on` on Morpho. It can't be turned `on` for Aave anymore because:

```
if (market.isCollateral) revert Errors.AssetIsCollateralOnMorpho();
```

But it also can't be turned `off` on Morpho anymore because of:

```
if (!_pool.getUserConfiguration(address(this)).isUsingAsCollateral(_pool.getReserveData(underlying).id) ↲
↪   )
↪   {
    revert Errors.AssetNotCollateralOnPool();
}
```

This will be bad if new users deposit after having withdrawn the entire asset. The asset is collateral on Morpho but not on Aave, breaking an important invariant that could lead to liquidating the Morpho Aave position.

Recommendation: Keep some buffer so not everything can be withdrawn, however, this doesn't completely fix the issue because liquidations allow seizing more assets than have been deposited. Consider always being able to turn an asset `off` on Morpho, regardless of the pool state. The check that it's also `on` on the pool should only be required when turning an asset `on` as collateral on Morpho.

Morpho:

- First, we plan to send aTokens for all assets to Morpho to prevent the LTV = 0 griefing attack so the only way it's possible is for Morpho to be liquidated.
- This scenario is very unlikely to happen and if Morpho gets liquidated the state is desync anyway.
- At the first deposit of a user or unmatching process, underlying will be deposited on the pool, resetting the asset as collateral by default even after the potential changes added by this PR on Aave.

Spearbit: Acknowledged.

5.3.2 First reward claim is zero for newly listed reward tokens

Severity: Medium Risk

Context: RewardsManager.sol#L291, RewardsManager.sol#L434, aave-v3/RewardsDistributor.sol#L261

Description: When Aave adds a new reward token for an asset, the reward index for this `(asset, reward)` pair starts at 0. When an update in Morpho's reward manager occurs, it initializes all rewards for the asset and would initialize this new reward token with a `startingIndex` of `0`.

1. Time passes and emissions accumulate to all pool users, resulting in a new index `assetIndex`. Users who deposited on the pool through Morpho before this reward token was listed should receive their fair share of the entire emission rewards `(assetIndex - 0) * oldBalance` but they currently receive zero because `getRewards` returns early if the user's computed index is `0`.
2. Also note that the external `getUserAssetIndex(address user, address asset, address reward)` can be inaccurate because it doesn't simulate setting the `startingIndex` for reward tokens that haven't been set yet.
3. A smaller issue that can happen when new reward tokens are added is that updates to the `startingIndex` are late, the `startingIndex` isn't initialized to `0` but to some asset index that accrued emissions for some time. Morpho on-pool users would lose some rewards until *the first update* to the asset. (They should accrue from index 0 but accrue from `startingIndex`.) Given frequent calls to the RewardManager that initializes all rewards for an asset, this difference should be negligible.

Recommendation: The special case for a *computed* user index (`_computeUserIndex`) of `0` in `getRewards` seems unnecessary because initializing `startingIndex` is always done before calling `_computeUserIndex` (except for the external `getUserAssetIndex` function). With the way `userIndex` is chosen in `_computeUserIndex`, a `userIndex` of `0` means `localRewardData.startingIndex` is `0`, i.e., the `(asset, reward)` pair was correctly *initialized to 0* in the contract.

```
function _getRewards(uint256 userBalance, uint256 reserveIndex, uint256 userIndex, uint256 assetUnit)
    internal
    pure
    returns (uint256 rewards)
{
-   // If `userIndex` is 0, it means that it has not accrued reward yet.
-   if (userIndex == 0) return 0;

    rewards = userBalance * (reserveIndex - userIndex);
    assembly {
        rewards := div(rewards, assetUnit)
    }
}
```

Morpho: The suggested fix was implemented in PR 791.

However, it was also suggested to update the `getUserAssetIndex` getter, which does not take into account a potential starting `index != 0`. I disagree with this, because exposing a virtual starting index update is inaccurate too: it's not guaranteed that the starting index of a reward asset already listed but not tracked by Morpho would be updated in the same block as the query

A first version of such a change was drafted in PR 795 but I don't think it's more accurate, so I'd rather be in favor of keeping the current version of the RewardsManager, acknowledging that the user index exposed through the getter may just not reflect a potential starting index update happening in the same block

Spearbit: Marked as fixed by PR 791. It would be more accurate (it would even be perfectly accurate from a third-party smart contract's POV calling it) but agree that this is an edge case that shouldn't happen often and it's unclear what this view function is even used for and if it needs to be fully accurate. The gain might not be worth the more complex code flow that is required for the fix. Documenting this limitation for third parties might be enough.

5.3.3 Disable creating markets for siloed assets

Severity: Medium Risk

Context: aave-v3/UserConfiguration.sol#L214

Description: Aave-v3 introduced siloed-borrow assets and siloed-borrow mode for users

> *"This feature allow assets with potentially manipulatable oracles (for example illiquid Uni V3 pairs) to be listed on Aave as single borrow asset i.e. if user borrows siloed asset, they cannot borrow any other asset. This helps mitigating the risk associated with such assets from impacting the overall solvency of the protocol."* - Aave Docs

The Morpho contract should *not* be in siloed-borrowing mode to avoid its restrictions on borrowing any other listed assets, especially as borrowing on the pool might be required for withdrawals. If a market for the siloed asset is created at deployment, users might borrow the siloed asset and break borrowing any of the other assets.

Recommendation: There are two possible ways of handling siloed assets:

- Disabling market creations

Disallow creating markets for siloed assets as they shouldn't be borrowed on Morpho. Using them as collateral is also useless as they will likely have an LT/LTV of 0 on Aave.

> "*A user can supply* Siloed Asset *just like any other asset using* supply() *method in* `pool.sol`*, though, the asset will not be enabled to use as collateral i.e. supplied amount will not add to the total collateral balance of the user."* - Aave Docs

The Aave docs are misleading as supplying the siloed assets does indeed enable them as collateral and there are no further restrictions on a siloed asset's LT/LTV. However, as this asset class is intended for assets with "potentially manipulatable oracles", using them as collateral should be disabled on Morpho.

- Deploying a custom Morpho pool

Alternatively, if there is high demand for users to borrow this asset, Morpho can deploy a new set of contracts, similar to what is being done for the efficiency mode. The only suppliable and borrowable asset should be the siloed asset, all other markets should only allow supplying and withdrawing as collateral.

Morpho: Fixed in PR 633.

Spearbit: Fixed. The `_createMarket` function now reverts when listing siloed assets. Setting an asset as siloed borrowing while it is already listed is unlikely to occur and even if it occurs there are likely already other borrow positions on Morpho and borrowing the siloed asset would fail.

5.3.4 A high value of `_defaultIterations` could make the withdrawal and repay operations revert because of OOG

Severity: Medium Risk

Context: PositionsManager.sol#L146-L147, PositionsManager.sol#L176-L178, MatchingEngine.sol#L128-L158

Description: When the user executes some actions, he/she can specify their own `maxIterations` parameter. The user `maxIterations` parameter is directly used in `supplyLogic` and `borrowLogic`.

In the `withdrawLogic` Morpho is recalculating the `maxIterations` to be used internally as `Math.max(_defaultIterations.withdraw, maxIterations)` and in `repayLogic` is directly using `_defaultIterations.repay` as the max number of iterations.

This parameter is used as the maximum number of iterations that the matching engine can do to match suppliers/borrowers during promotion/demotion operations.

```
function _promoteOrDemote(
    LogarithmicBuckets.Buckets storage poolBuckets,
    LogarithmicBuckets.Buckets storage p2pBuckets,
    Types.MatchingEngineVars memory vars
) internal returns (uint256 processed, uint256 iterationsDone) {
    if (vars.maxIterations == 0) return (0, 0);

    uint256 remaining = vars.amount;

    // matching engine code...

    for (; iterationsDone < vars.maxIterations && remaining != 0; ++iterationsDone) {
        // matching engine code

        (onPool, inP2P, remaining) =
            vars.step(...);

        // matching engine code...
    }

    // matching engine code...
}
```

As you can see, the iteration keeps going on until the matching engine has matched enough balance or the iterations have reached the maximum number of iterations.

If the matching engine cannot match enough balance, it could revert because of OOG if `vars.maxIterations` is a high value. For the supply or borrow operations, the user is responsible for the specified number of iterations that might be done during the matching process, in that case, if the operations revert because of OGG, it's not an issue per se.

The problem arises for withdraw and replay operations where Morpho is forcing the number of operations and could make all those transactions always revert in case the matching engine does not match enough balance in time. Keep in mind that even if the transaction does not revert during the `_promoteOrDemote` logic, it could revert during the following operations just because the `_promoteOrDemote` has consumed enough gas to make the following operations to use the remaining gas.

Recommendation: Consider stress testing the correct value to be used for `_defaultIterations.repay` and `_defaultIterations.withdraw` to prevent those operations to revert because of OOG.

Morpho: We're conducting studies on the matching efficiency given a max iterations as well as the gas consumed. This study should give us the appropriate max iterations that we should set.

Spearbit: Acknowledged.

5.3.5 `Morpho` should check that the `_positionsManager` used has the same `_E_MODE_CATEGORY_ID` and `_ADDRESSES_PROVIDER` values used by the `Morpho` contract itself

Severity: Medium Risk

Context: Morpho.sol#L48, MorphoSetters.sol#L59

Description: Because `_E_MODE_CATEGORY_ID` and `_ADDRESSES_PROVIDER` are `immutable` variables and because `Morpho` is calling the `PositionsManager` in a `delegatecall` context, it's fundamental that both `Morpho` and `PositionsManager` have been initialized with the same `_E_MODE_CATEGORY_ID` and `_ADDRESSES_PROVIDER` values.

Morpho should also check the value of the `PositionsManager._E_MODE_CATEGORY_ID` and `PositionsManager._ADDRESSES_PROVIDER` in both the `setPositionsManager` and `initialize` function.

Recommendation: Implement in both the `setPositionsManager` and `initialize` function a check that reverts if the value inside `_E_MODE_CATEGORY_ID` and `_ADDRESSES_PROVIDER` `Morpho` and `PositionsManager` are not equal.

Note that Morpho has to create a public getter for those immutable values because they are declared as internal.

Morpho: We decided for putting variables in storage. We know it's not ideal in terms of gas but we prefer to do so for safety considerations.

Spearbit: The Morpho team has decided to implement the recommendation by changing the _ADDRESSES_-PROVIDER, _POOL, and _E_MODE_CATEGORY_ID variables from immutable to storage. By doing that, Morpho contract will not rely on the immutable values stored in the PositionManager when called via delegatecall but will instead use the storage value of the Morpho's contract.

The implementation has been done in PR 597.

Two additional considerations to be noted after the PR changes:

- Morpho has removed the event Events.EModeSet that was emitted during the Morpho.initialize function. They are now relying on the Aave side event emission. They should remember to update their current monitoring system to switch to the new behavior.
- RewardsManager constructor does not include the _pool in the list of user's inputs and relies on the values that come from IMorpho(morpho).pool(). With the new behavior of the PR the Morpho contract could have been deployed but not initialized yet, in that case IMorpho(morpho).pool() would return address(0). Morpho should consider adding this additional check inside the constructor.

5.3.6 In _authorizeLiquidate, when healthFactor is equal to Constants.DEFAULT_LIQUIDATION_THRESHOLD Morpho is wrongly setting close factor to DEFAULT_CLOSE_FACTOR

Severity: Medium Risk

Context: PositionsManagerInternal.sol#L181-L190

Description: When the borrower's healthFactor is equal to Constants.MIN_LIQUIDATION_THRESHOLD Morpho is returning the wrong value for the closeFactor allowing only liquidate 50% of the collateral instead of the whole amount.

When the healthFactor is lower or equal to the Constants.MIN_LIQUIDATION_THRESHOLD Morpho should return Constants.MAX_CLOSE_FACTOR following the same logic applied by Aave.

Note that the user cannot be liquidated even if healthFactor == MIN_LIQUIDATION_THRESHOLD if the priceOr-acleSentinel is set and IPriceOracleSentinel(params.priceOracleSentinel).isLiquidationAllowed() == false. See how Aave performs the check inside validateLiquidationCall.

Recommendation: Consider decoupling the logic that checks if a user can be liquidated and the logic that calculates the correct close factor to be used for the liquidation. After doing that, apply the correct fixes to follow the same behavior of Aave to determine the close factor.

Morpho: Recommendation implemented in PR 571.

Spearbit: Fixed.

5.3.7 _authorizeBorrow does not check if the Aave price oracle sentinel allows the borrowing operation

Severity: Medium Risk

Context: PositionsManagerInternal.sol#L106-L126

Description: Inside the Aave validation logic for the borrow operation, there's an additional check that prevents the user from performing the operation if it has been not allowed inside the priceOracleSentinel

```
require(
  params.priceOracleSentinel == address(0) ||
    IPriceOracleSentinel(params.priceOracleSentinel).isBorrowAllowed(),
  Errors.PRICE_ORACLE_SENTINEL_CHECK_FAILED
);
```

Morpho should implement the same check. If for any reason the borrow operation has been disabled on Aave, it should also be disabled on Morpho itself. While the transaction would fail in case Morpho's user would need to perform the borrow on the pool, there could be cases where the user is completely matched in P2P. In those cases, the user would have performed a borrow even if the borrow operation was not allowed on the underlying Aave pool.

Recommendation: Implement the `priceOracleSentinel` check, reverting in case `IPriceOracleSentinel(priceOracleSentinel).isBorrowAllowed() == false`.

Morpho: The recommendation has been implemented in PR 599.

Spearbit: Fixed.

5.4 Low Risk

5.4.1 `_updateInDS` does not "bubble up" the updated values of `onPool` and `inP2P`

Severity: Low Risk

Context: MorphoInternal.sol#L352-L353, MorphoInternal.sol#L410

Description: The `_updateInDS` function takes as input `uint256 onPool` and `uint256 inP2P` that are passed not as reference, but as pure values.

```
function _updateInDS(
    address poolToken,
    address user,
    LogarithmicBuckets.Buckets storage poolBuckets,
    LogarithmicBuckets.Buckets storage p2pBuckets,
    uint256 onPool,
    uint256 inP2P,
    bool demoting
) internal {

    if (onPool <= Constants.DUST_THRESHOLD) onPool = 0;
    if (inP2P <= Constants.DUST_THRESHOLD) inP2P = 0;

    // ... other logic of the function
}
```

Those values, if lower or equal to `Constants.DUST_THRESHOLD` will be set to `0`. The issue is that the updated version of `onPool` and `inP2P` is never bubbled up to the original caller that will later use those values that could have been changed by the `_updateInDS` logic.

For example, the `_updateBorrowerInDS` function call `_updateInDS` and relies on the value of `onPool` and `inP2P` to understand if the user should be removed or added to the list of borrowers.

```
function _updateBorrowerInDS(address underlying, address user, uint256 onPool, uint256 inP2P, bool
↪   demoting) internal {
    _updateInDS(
        _market[underlying].variableDebtToken,
        user,
        _marketBalances[underlying].poolBorrowers,
        _marketBalances[underlying].p2pBorrowers,
        onPool,
        inP2P,
        demoting
    );

    if (onPool == 0 && inP2P == 0) _userBorrows[user].remove(underlying);
    else _userBorrows[user].add(underlying);
}
```

Let's assume that `inP2P` and `onPool` passed as `_updateBorrowerInDS` inputs were equal to `1` (the value of `DUST_-THRESHOLD`).

In this case, `_updateInDS` would update those values to zero because `1 <= DUST_THRESHOLD` and would remove the user from both the `poolBucket` and `p2pBuckets` of the `underlying`.

When then the function returns in the `_updateBorrowerInDS` context, the same user would not remove the `under-lying` from his/her `_userBorrows` list of assets because the updated values of `onPool` and `inP2P` have not been bubbled up by the `_updateInDS` function.

The same conclusion could be made for all the "root" level codes that rely on the `onPool` and `inP2P` values that could not have been updated with the new `0` value set by `_updateInDS`.

Recommendation: Consider "bubbling up" the updated value of both `onPool` and `inP2P` variables from `_up-dateInDS` to be able to use that updated values also in the "root" level of the code.

Morpho: Recommendation implemented in PR 626.

Spearbit: Fixed.

5.4.2 There is no guarantee that the `_rewardsManager` is set when calling `claimRewards`

Severity: Low Risk

Context: Morpho.sol#L268

Description: Since the `_rewardsManager` address is set using a setter function in Morpho only and not in the `MorphoStorage.sol` constructor there is no guarantee that the `_rewardsManager` is not the default address(0) value. This could cause failures when calling `claimRewards` if Morpho forgets to set the `_rewardsManager`.

Recommendation: When calling `claimRewards` there should be a check that the `rewardsManager` is not `ad-dress(0)`.

Morpho: Fixed in PR 658.

Spearbit: Fixed.

5.4.3 Its Impossible to set `_isClaimRewardsPaused`

Severity: Low Risk

Context: Morpho.sol#L266

Description: The `claimRewards` function checks the `isClaimRewardsPaused` boolean value and reverts if it is true. Currently, there is no setter function in the code base that sets the `_isClaimRewardsPaused` boolean so it is impossible to change.

Recommendation: Add a setter function with the `onlyOwner` modifier that is able to set the `_isClaimRewarsPaused` value in storage.

Morpho: Fixed in PR 567.

Spearbit: Fixed.

5.4.4 User rewards can be claimed to treasury by DAO

Severity: Low Risk

Context: Morpho.sol#L269, MorphoInternal.sol#L112

Description: When a user claims rewards, the rewards for the entire Morpho contract position on Aave are claimed. The excess rewards remain in the Morpho contract for until all users claimed their rewards. These rewards are not tracked and can be withdrawn by the DAO through a `claimToTreasury` call.

Recommendation: Consider tracking the reward balance or pay attention when claiming fees to the treasury by setting the `amounts` parameter only to the accrued fees.

Morpho: Yes, I don't think we'll add more logic for that since it would add lots of logic in the `claimToTreasury` function.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 decreaseDelta lib function should return early if amount == 0

Severity: Gas Optimization

Context: MarketSideDeltaLib.sol#L55

Description: The passed in amount should be checked for a zero value, and in that condition, return early from the function. The way it currently is unnecessarily consumes more gas, and emits change events that for values that don't end up changing (`newScaledDelta`).

Checking for `amount == 0` is already being done in the increaseDelta function.

Recommendation: Add a check `if (amount == 0) return` at the top of the `decreaseDelta` function.

Morpho: Fixed in PR 600.

Spearbit: Fixed.

5.5.2 Smaller gas optimizations

Severity: Gas Optimization

Context: See below

Description: There are several small expressions that can be further gas optimized.

Recommendation: Consider changing these:

- MarketLib.sol#L219: The subtraction can be unchecked because of the `if` in the previous line.
- MarketLib.sol#244: An unchecked subtraction can be used instead of `zeroFloorSub` because of `matchedIdle = Math.min(idleSupply, amount)` being at most `idleSupply`.
- MorphoInternal.sol#L294: Consider using a less gas intensive function that just retrieves `underlyingPrice` and `tokenUnit` without having to execute the rest of the `_assetLiquidityData` that is not useful in this function's context. That function could then be called internally by `_assetLiquidityData` and enhanced with the additional values needed.
- MarketLib.sol#L209: consider postponing the `mul` operation after the check on `reserve.configuration.getSupplyCap() == 0` value

Morpho: Fixed in 654.

Spearbit: Fixed.

5.5.3 Gas: Optimize `LogarithmicBuckets.getMatch`

Severity: Gas Optimization

Context: LogarithmicBuckets.sol#L72

Description: The `getMatch` function of the logarithmic bucket first checks for a bucket that is the next higher bucket than the bucket the provided value would be in. If no higher bucket is found it searches for a bucket that is the highest bucket that "is in both `bucketsMask` and `lowerMask`." However, we already know that any bucket we can now find will be in `lowerMask` as `lowerMask` is the mask corresponding to all buckets less than or equal to `value`'s bucket. Instead, we can just directly look for the highest bucket in `bucketsMask`.

Recommendation: Remove the `lowerMask` check and just return the highest bucket in the second step:

```
function getMatch(Buckets storage _buckets, uint256 _value) internal view returns (address) {
    uint256 bucketsMask = _buckets.bucketsMask;
    if (bucketsMask == 0) return address(0);
    uint256 lowerMask = setLowerBits(_value);

    uint256 next = nextBucket(lowerMask, bucketsMask);

    if (next != 0) return _buckets.buckets[next].getHead();

    uint256 prev = highestBucket(bucketsMask);

    return _buckets.buckets[prev].getHead();
}

function highestBucket(uint256 bucketsMask) internal pure returns (uint256) {
    uint256 lowerBucketsMask = setLowerBits(bucketsMask);
    return lowerBucketsMask ^ (lowerBucketsMask >> 1);
}
```

Morpho: Fixed in PR 34.

Spearbit: Fixed.

5.6 Informational

5.6.1 Consider reverting the `supplyCollateralLogic` execution when `amount.rayDivDown(poolSupplyIndex)` is equal to zero

Severity: Informational

Context: PositionsManagerInternal.sol#L411-L426, PositionsManagerInternal.sol#L512

Description: In Aave, when an `AToken/VariableDebtToken` is minted or burned, the transaction will revert if the `amount` divided by the `index` is equal to zero. You can see the check in the implementation of `_mintScaled` and `_burnScaled` functions in the Aave codebase.

Morpho, with PR 688, has decided to prevent supply to the pool in this scenario to avoid a revert of the operation.

Before the PR, if the user had supplied an `amount` for which `amount.rayDivDown(poolSupplyIndex)` would be equal to zero, the operation would have reverted at the Aave level during the mint operation of the AToken. With the PR, the operation will proceed because the supply to the Aave pool is skipped (see `PoolLib.supplyToPool`).

Allowing this scenario in this specific context for the `supplyCollateralLogic` function will bring the following side effects:

- The supplied user's `amount` will remain in Morpho's contract and will not be supplied to the Aave pool.
- The user's accounting system is not updated because `collateralBalance` is increased by `amount.rayDivDown(poolSupplyIndex)` which is equal to zero.

- If the `marketBalances.collateral[onBehalf]` was equal to zero (the user has never supplied the `underlying` to Morpho) the `underlying` token would be wrongly added to the `_userCollaterals[onBehalf]` storage, even if the amount supplied to Morpho (and to Aave) is equal to zero.
- The user will not be able to withdraw the provided `amount` because the amount has not been accounted for in the storage.
- `Events.CollateralSupplied` event is emitted even if the `amount` (used as an event parameter) has not been accounted to the user.

Recommendation: Consider reverting the `supplyCollateral` and `supplyCollateralWithPermit` operations when `amount.rayDivDown(poolSupplyIndex)` is equal to zero.

Morpho: We won't change the current code for the following reasons:

- Rounding down allows to do the accounting in favor of the protocol instead of the user which is the safer option.
- The error is bounded by 1 WEI of scaled balance and computations are already not precise to the WEI (even for scaled balances). Index being in the order of magnitudes of 1-10 RAY, the final error is bounded by 10 wei.
- It's a bit annoying that the market would be added to the list of collaterals of the user but I'd say that the user can easily prevent that by not supplying negligible amounts to Morpho (which do not make sense for a user anyway).

Spearbit: Acknowledged.

5.6.2 `WETHGateway` does not validate the `constructor`'s input parameters

Severity: Informational

Context: WETHGateway.sol#L31

Description: The current implementation of the `WETHGateway` contracts does not validate the user's parameters during the `constructor`. In this specific case, the `constructor` should revert if `morpho` address is equal to `address(0)`.

Recommendation: Morpho should add sanity checks related to the user's input parameters, preventing the contract from being successfully created if they do not pass the validation.

In this specific case, the `constructor` should revert if `morpho` address is equal to `address(0)`.

Morpho: Recommendation implemented in PR 601.

Spearbit: Verified.

5.6.3 Missing/wrong natspec, typos, minor refactors and renaming of variables to be more meaningful

Severity: Informational

Context: See below

Description: In general, the current codebase does not cover all the functions, events, structs, or state variables with proper natspec.

Below you can find a list of small specific improvements regarding typos, missing/wrong natspec, or suggestions to rename variables to a more meaningful/correct name

- RewardsManager.sol#L28: consider renaming the `balance` variable in `UserAssetBalance` to `scaledBalance`
- PositionsManagerInternal.sol#L289-L297, PositionsManagerInternal.sol#L352-L362: consider better documenting this part of the code because at first sight it's not crystal clear why the code is structured in this way. For more context, see the PR comment in the spearbit audit repo linked to it.

- MorphoInternal.sol#L469-L521: consider moving the `_calculateAmountToSeize` function from `MorphoInternal` to `PositionsManagerInternal` contract. This function is only used internally by the `PositionsManagerInternal`. Note that there could be more instances of these kinds of "refactoring" of the code inside other contracts.

Recommendation: Consider carefully updating/adding natspec for each function, event, struct, custom error, and state variable to have complete documentation of your code. This operation will be beneficial for Morpho developers, integrators and auditors.

Consider also implementing the suggestions regarding the issues found in the current natspec or variable renaming.

Morpho: Part of the recommendations have been implemented in the PR 621.

Spearbit: The Morpho team has stated that all the missing/wrong natspec comments will be handled in a different PR. The Morpho team should consider reviewing the codebase to find other possible `function` that could be "bubbled up" to have a more clean codebase.

5.6.4 No validation checks on the `newDefaultIterations` struct

Severity: Informational

Context: Morpho.sol#L49

Description: The `initialize` function takes in a `newDefaultIterations` struct and does not perform validation for any of its fields.

Recommendation: Consider validating at least a max value to avoid undesirable behavior if no matches occur in time.

Morpho: We won't implement it. It's up to the governance to set relevant values. If Morpho is wrongly initialized it can still be updated later and it's not clear what would be a correct "max value" either.

Spearbit: Acknowledged.

5.6.5 No validation check for `newPositionsManager` address

Severity: Informational

Context: Morpho.sol#L48

Description: The `initialize` function does not ensure that the `newPositionsManager` is not a 0 address.

Recommendation: Proper validation should be added to the `newPositionsManager` argument and the function should revert if it is indeed a 0 address.

Morpho: We don't think it's worth it. Either the DAO can redeploy another instance of Morpho or set the positionsManager to the correct address later on using the setter.

Spearbit: Acknowledged.

5.6.6 Missing Natspec function documentation

Severity: Informational

Context: PositionsManager.sol#L128-L132

Description: The `repayLogic` function currently has Natspec documentation for every function argument except for the `repayer` argument.

Recommendation: Natspec documentation should be added for the `repayer` function argument on the `repay-Logic` function.

Morpho: Fixed in PR 634.

Spearbit: Fixed.

5.6.7 `approveManagerWithSig` user experience could be improved

Severity: Informational

Context: Morpho.sol#L250

Description: With the current implementation of the `approveManagerWithSig` signers must wait that the previous signers have consumed the nonce to be able to call `approveManagerWithSig`.

Inside the function, there's a specific check that will revert if the signature has been signed with a `nonce` that is not equal to the current one assigned to the `delegator`, this means that signatures that use "future" nonce will not be able to be approved until previous nonce has been consumed.

```
uint256 usedNonce = _userNonce[signatory]++;
if (nonce != usedNonce) revert Errors.InvalidNonce();
```

Let's make an example: `delegator` want to allow 2 managers via signature

1) Generate `sig_0` for `manager1` with `nonce_0`.
2) Generate `sig_1` for `manager2` with `nonce_1`.
3) If no-one executes `approveManagerWithSig(sig_0)` the `sig_1` (and all the signatures based on incremented nonces) cannot be executed. It's true that at some point someone/the signer will execute it.

Recommendation: If this is an acceptable behavior, consider documenting it and explaining to the `delegator` which are the possible limitation of the process.

Morpho: I believe that this should be considered intended behavior. Alternative behaviors create uncertainty as to what order signatures would be consumed, and this could be problematic if, for example, a user has signatures for both approving and unapproving a manager.

Spearbit: Acknowledged.

5.6.8 Consider hardcoding `msg.sender` as the `from` parameter for certain actions

Severity: Informational

Context: Morpho.sol#L283

Description: The following Morpho functions have a `from` field that is always set to `msg.sender`: `_supply`, `_supplyCollateral`, `_repay` (and `liquidate`'s liquidator). Generally, a parameter that is always set to the same value does not need to be a parameter and can be hardcoded further down the call graph. However, this might make the code easier to understand in certain circumstances. In this case, we believe that removing the mentioned parameters and directly using `msg.sender` in the delegate-called `PositionManager *Logic` functions makes the code easier to reason about. The `from` parameter has important security considerations as it is used as the `owner` in an `ERC20.transferFrom` call and, unlike `borrow` or `withdraw`, does not validate this parameter with a `_validatePermission` call. Therefore these functions are only secure when called with `from=msg.sender` and removing the parameter avoids having to analyze all call sites. Additionally, it will lead to gas improvements as the calldata is reduced.

Morpho: The "from" parameter is quite useful if we want to implement something like meta transactions in the future which is why we added it. We are not including this in the initial release, but we have talked about it internally at length and I don't think it's much extra gas to have this extra calldata that would make this potential feature much easier to implement.

Spearbit: Acknowledged.

5.6.9 Missing user markets check when liquidating

Severity: Informational

Context: PositionsManager.sol#L238

Description: The liquidation does not check if the user who gets liquidated actually joined the collateral and borrow markets.

Recommendation: Instead of setting the repayment amount to zero and continuing with the zero liquidation, consider reverting if the user is not in the markets.

Morpho: `liquidateLogic` now reverts for both the repay and seized amount of zero. fixed here PR 629.

Spearbit: Fixed.

5.6.10 Consider reverting instead of returning zero inside `repayLogic`, `withdrawLogic`, `withdrawCollateralLogic` and `liquidateLogic` function

Severity: Informational

Context: PositionsManager.sol#L142, PositionsManager.sol#L174, PositionsManager.sol#L204, PositionsManager.sol#L249

Description: Position manager always checks the user inputs via different validation functions. One of the validations is that the input's `amount` must be greater than zero, otherwise, the transaction reverts with `revert Errors.AmountIsZero()`.

The same behavior is not followed in those cases where the re-calculated amount is still zero. For example, in `repayLogic` after re-calculating the max amount that can be repaid by executing

```
amount = Math.min(_getUserBorrowBalanceFromIndexes(underlying, onBehalf, indexes), amount);
```

In this case, Morpho simply executes `if (amount == 0) return 0;`

Note that `liquidateLogic` should be handled differently because both the borrow amount and/or the collateral amount could be equal to zero. In this case, it would be better to revert with a different custom error based on which of the two amounts are equal to zero.

Recommendation: Consider reverting, with a specific error code for each case, even when the re-calculated amount is equal to zero instead of simply returning zero.

Morpho: Recommendation implemented in PR 629.

Spearbit: Fixed.

5.6.11 `PERMIT2` operations like `transferFrom2` and `simplePermit2` will revert if `amount` is greater than `type(uint160).max`

Severity: Informational

Context: Morpho.sol#L90-L92, Morpho.sol#L121-L123, Morpho.sol#L166-L168, PositionsManager.sol#L62, PositionsManager.sol#L87, PositionsManager.sol#L144, PositionsManager.sol#L251

Description: Both `Morpho.sol` and `PositionsManager.sol` uses the Permit2 lib. The current implementation of the `permit2` lib explicitly restricts the amount of token to `uint160` by calling `amount.toUint160()`

On Morpho, the amount is expressed as a `uint256` and the user could, in theory, pass an amount that is greater than `type(uint160).max`. By doing so, the transaction would revert when it interacts with the permit2 lib.

Recommendation: Consider restricting the `amount` to `uint160` or document the behavior as a natspec comment and on the documentation visible to the users and integrators.

Morpho: I don't think a fix for this is necessary as long as the transaction reverts properly.

Spearbit: Acknowledged.

5.6.12 Both `_wrapETH` and `_unwrapAndTransferETH` do not check if the `amount` is zero

Severity: Informational

Context: WETHGateway.sol#L95, WETHGateway.sol#L100-L101

Description: Both `_wrapETH` and `_unwrapAndTransferETH` are not checking if the amount `amount` of tokens is greater than zero. If the amount is equal to zero, Morpho should avoid making the external call or simply revert.

Recommendation: Avoid execution of the external call to the WETH contract or revert if the amount is equal to zero. Morpho should anyway consider monitoring these events because each interaction with Morpho should always return an amount of token greater than zero.

Morpho: The recommendation has been implemented in PR 655.

Spearbit: Fixed. Morpho has decided to `revert` when the amount wrapped/unwrapped is equal to zero. In the integrator docs, Morpho should make the integrators aware of this behavior for the `borrowETH`, `withdrawETH`, and `withdrawCollateralETH`. In those specific cases, the integrators have to handle the possible revert with a `try/catch` to not make the whole tx fail.

5.6.13 Document further contraints on `BucketDLL`'s `insert` and `remove` functions

Severity: Informational

Context: BucketDLL.sol#L49, BucketDLL.sol#L67

Description: Besides the constraint that `id` may not be zero, there are further constraints that are required for the `insert` and `remove` functions to work correctly:

- `insert`: "This function should not be called with an `_id` that is already in the list." Otherwise, it would overwrite the existing `_id`.
- `remove`: "This function should not be called with an `_id` that is not in the list." Otherwise, it would set all of `_list.accounts[0]` to `address(0)`, i.e., mark the list as empty.

Recommendation: Consider adding these two constraints as `@dev` comments next to the "This function should not be called with `_id` equal to address 0." constraint.

```
/// @dev This function should not be called with `_id` equal to address 0.
+ /// @dev This function should not be called with an `_id` that is not in the list.
function remove(List storage _list, address _id) internal returns (bool) {
    ...
}

/// @dev This function should not be called with `_id` equal to address 0.
+ /// @dev This function should not be called with an `_id` that is already in the list.
function insert(
    List storage _list,
    address _id,
    bool _head
) internal returns (bool) {
    ...
}
```

Morpho: Addressed in PR 34.

Spearbit: Fixed.