Morpho Security Review

Reviewers

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Security Researcher

Jay Jonah, Security Researcher

hack3r-0m, Security Researcher

Report prepared by: Pablo Misirov

August 24, 2022

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 https://spearbit.com.

2 Introduction

Morpho is a new gateway to decentralized lending. It is a Peer-to-Peer (P2P) layer on top of lending pools like Compound or Aave. Morpho is a lending pool optimizer: it improves the capital efficiency of positions on lending pools by seamlessly matching lenders and borrowers peer-to-peer. As such, Morpho improves your rates while preserving the same experience, the same liquidity and the same parameters (collateral factors, oracles etc) as the underlying pool. This means that, using Morpho, you either receive an improved,P2P APY or, in the worst-case scenario, retain the APY of the underlying pool.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Morpho according to the specific commit by a three person team. 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 14 days in total, Morpho engaged with Spearbit to review morpho-contracts. In this period of time a total of 41 issues were found.

Update 24/08/2022: As per client's request, it is stated and must be assumed that the audited codebase herein pertains to the beta version of the product and not the final code released to production.

Summary

Project Name	Morpho
Repository	morpho-contracts
Commit	a861d50597ae047640727...
Type of Project	P2P lending, DeFi
Audit Timeline	March 15 - 29, 2022
Methods	Manual Review

Issues Found

Critical Risk	3
High Risk	7
Medium Risk	11
Low Risk	9
Gas Optimizations	4
Informational	7
Total Issues	41

5 Findings

5.1 Critical Risk

5.1.1 Wrong P2P exchange rate calculation

Severity: *Critical Risk*

Context: MarketsManagerForAave.sol#L436

Description: `_p2pDelta` is divided by `_poolIndex` and multiplied by `_p2pRate`, nevertheless it should have been multiplied by `_poolIndex` and divided by `_p2pRate` to compute the correct share of the delta. This leads to wrong P2P rates throughout all markets if supply / borrow delta is involved.

Recommendation: Change order and adjust return values accordingly.

```
 uint256 shareOfTheDelta = _p2pDelta
             .wadToRay()
-            .rayMul(_p2pRate)
-            .rayDiv(_poolIndex)
+            .rayMul(_poolIndex)
+            .rayDiv(_p2pRate)
             .rayDiv(_p2pAmount.wadToRay());
```

Morpho: Fixed in PR #536, `_computeNewP2PExchangeRate` is changed as recommended.

Spearbit: Acknowledged.

5.1.2 MatchingEngineForAave is using the wrong totalSupply in updateBorrowers

Severity: *Critical Risk*

Context: MatchingEngineForAave.sol#L376-L385

Description: `_poolTokenAddress` is referencing `AToken` so the `totalStaked` would be the total supply of the `AToken`.

In this case, the `totalStaked` should reference the total supply of the `DebtToken`, otherwise the user would be rewarded for a wrong amount of reward.

Recommendation: Use the correct token address to query `scaledTotalSupply` as follows:

```
address variableDebtTokenAddress = lendingPool
.getReserveData(IAToken(_poolTokenAddress).UNDERLYING_ASSET_ADDRESS())
.variableDebtTokenAddress;
uint256 totalStaked = IScaledBalanceToken(variableDebtTokenAddress).scaledTotalSupply();
```

Spearbit: Fixed, recommendation was implemented in the PR #554

5.1.3 RewardsManagerAave does not verify token addresses

Severity: *Critical Risk*

Context: RewardsManagerForAave.sol#L145-L147

Description: Aave has 3 different types of tokens: aToken, stable debt token and variable debt token (a/s/vToken). Aave's incentive controller can define rewards for all of them but Morpho never uses a stable-rate borrows token (sToken). The public `accrueUserUnclaimedRewards` function allows passing arbitrary token addresses for which to accrue user rewards. Current code assumes that if the token is not the variable debt token, then it must be the aToken, and uses the user's supply balance for the reward calculation as follows:

```
uint256 stakedByUser = reserve.variableDebtTokenAddress == asset
    ? positionsManager.borrowBalanceInOf(reserve.aTokenAddress, _user).onPool
    : positionsManager.supplyBalanceInOf(reserve.aTokenAddress, _user).onPool;
```

An attacker can accrue rewards by passing in an sToken address and steal from the contract, i.e:

- Attacker supplies a large amount of tokens for which sToken rewards are defined.
- The aToken reward index is updated to the latest index but the sToken index is not initialized.
- Attacker calls `accrueUserUnclaimedRewards([sToken])`, which will compute the difference between the current Aave reward index and user's sToken index, then multiply it by their supply balance.
- The user accumulated rewards in `userUnclaimedRewards[user]` can be withdrawn by calling `PositionManager.claimRewards([sToken, ...])`.
- Attacker withdraws their supplied tokens again.

The abovementioned steps can be performed in one single transaction to steal unclaimed rewards from all Morpho positions.

Recommendation: Verify the token address to be either an aToken or vToken.

```
function accrueUserUnclaimedRewards(address[] calldata _assets, address _user)
{
   // ...
   for (uint256 i = 0; i < _assets.length; i++) {
       address asset = _assets[i];
       DataTypes.ReserveData memory reserve = lendingPool.getReserveData(
           IGetterUnderlyingAsset(asset).UNDERLYING_ASSET_ADDRESS()
       );

-      uint256 stakedByUser = reserve.variableDebtTokenAddress == asset
-          ? positionsManager.borrowBalanceInOf(reserve.aTokenAddress, _user).onPool
-          : positionsManager.supplyBalanceInOf(reserve.aTokenAddress, _user).onPool;
+      uint256 stakedByUser;
+      if (reserve.variableDebtTokenAddress == asset) {
+        stakedByUser = positionsManager.borrowBalanceInOf(reserve.aTokenAddress, _user).onPool;
+      } else {
+        require(reserve.aTokenAddress == asset, "invalid asset");
+        stakedByUser = positionsManager.supplyBalanceInOf(reserve.aTokenAddress, _user).onPool;
+      }
       // ...
   }
}
```

Morpho: Fixed, the recommendation has been implemented in PR #554

Spearbit: Acknowledged.

5.2 High Risk

5.2.1 FullMath requires overflow behavior

Severity: High Risk

Context: FullMath.sol#L2

Description: UniswapV3's `FullMath.sol` is copied and migrated from an old solidity version to version 0.8 which reverts on overflows but the old `FullMath` relies on the implicit overflow behavior. The current code will revert on overflows when it should not, breaking the `SwapManagerUniV3` contract.

Recommendation: Use the official FullMath.sol 0.8 branch that wraps the code in an unchecked statement. See #40.

Spearbit: Fixed, the Uniswap V3 branch is added as a dependency in the PR #550

5.2.2 Morpho's USDT mainnet market can end up in broken state

Severity: *High Risk*

Context: PositionsManagerForAaveLogic.sol#L502

Description: Note that USDT on Ethereum mainnet is non-standard and requires resetting the approval to zero (see USDT L199) before being able to change it again.

In `_repayERC20ToPool` , it could be that `_amount` is approved but then `_amount = Math.min(...)` only repays a smaller amount, meaning there remains a non-zero approval for Aave. Any further `_repayERC20ToPool/_supplyERC20ToPool` calls will then revert in the approve call. Users cannot interact with most functions of the Morpho USDT market anymore.

Example: Assume the attacker is first to borrow from the USDT market on Morpho.

- Attacker borrows 1000 USDT through Morpho from the Aave pool (and some other collateral to cover the debt).
- Attacker directly interacts with Aave to repay 1 USDT of debt for Aave's Morpho account position.
- Attacker attempts to repay 1000 USDT on Morpho. It will approve 1000 USDT but the contract's debt balance is only 999 and the `_amount = Math.min(_amount, variableDebtToken.scaledBalanceOf(address(this)).mulWadByRay(_normalizedVariableDebt)` computation will only repay 999. An approval of 1 USDT remains.
- The USDT market is broken as it reverts on supply / repay calls when trying to approve the new amount

Recommendation: In `_repayERC20ToPool/_supplyERC20ToPool` do a `safeApprove(0)` first before the approval or ensure that the exact approved amount is always transferred by Aave, resetting the allowance to zero this way.

Morpho: This is fixed (not in the PR closed). Now we approve only what will be paid to Aave and no more. So after repay the allowance will always be 0.

Spearbit: Acknowledged.

5.2.3 Wrong reserve factor computation on P2P rates

Severity: *High Risk*

Context: MarketsManagerForAave.sol#L413-L418

Description: The reserve factor is taken on the entire P2P supply and borrow rates instead of just on the spread of the pool rates. It's currently overcharging suppliers and borrowers and making it possible to earn a worse rate on Morpho than the pool rates.

```
supplyP2PSPY[_marketAddress] =
    (meanSPY * (MAX_BASIS_POINTS - reserveFactor[_marketAddress])) /
    MAX_BASIS_POINTS;
borrowP2PSPY[_marketAddress] =
    (meanSPY * (MAX_BASIS_POINTS + reserveFactor[_marketAddress])) /
    MAX_BASIS_POINTS;
```

Recommendation: Fix the computation.

Morpho: The real reserve factor should apply only on the spread so you're right that this formula is wrong and needs to be updated. a + (1/2 +- f)(b-a) where f is the reserve factor.

Spearbit: Acknowledged, fixed in PR #565.

5.2.4 SwapManager assumes Morpho token is token0 of every token pair

Severity: *High Risk*

Context: SwapManagerUniV2.sol#L106

Description: The `consult` function wrongly assumes that the Morpho token is always the first token (`token0`) in the `Morpho <> Reward token` token pair. This could lead to inverted prices and a denial of service attack when claiming rewards as the wrongly calculated expected amount slippage check reverts.

Recommendation: Consider using similar code to the example UniswapV2 oracle. Note that depending on how this issue is fixed in `consult`, the caller of this function needs to be adjusted as well to return a Morpho token amount as `amountOut`.

Morpho: Fixed in PR #585.

Spearbit: Acknowledged.

5.2.5 SwapManager fails at updating TWAP

Severity: *High Risk*

Context: SwapManagerUniV2.sol#L83-L85

Description: The `update` function returns early without updating the TWAP if the elapsed time is past the TWAP period. Meaning, once the TWAP period passed the TWAP is stale and forever represents an old value. This could lead to a denial of service attack when claiming rewards as the wrongly calculated expected amount slippage check reverts.

Recommendation: Fix the code:

```
// ensure that at least one full period has passed since the last update
- if (timeElapsed >= PERIOD) {
+ if (timeElapsed < PERIOD) {
    return;
 }
```

Morpho: Fixed in PR #550

Spearbit: Acknowledged.

5.2.6 P2P rate can be manipulated as it's a lazy-updated snapshot

Severity: *High Risk*

Context: MarketsManagerForAave.sol#L408-L411

Description: The P2P rate is lazy-updated upon interactions with the Morpho protocol. It takes the mid-rate of the current Aave supply and borrow rate. It's possible to manipulate these rates before triggering an update on Morpho.

```
function _updateSPYs(address _marketAddress) internal {
    DataTypes.ReserveData memory reserveData = lendingPool.getReserveData(
        IAToken(_marketAddress).UNDERLYING_ASSET_ADDRESS()
    );

    uint256 meanSPY = Math.average(
        reserveData.currentLiquidityRate,
        reserveData.currentVariableBorrowRate
    ) / SECONDS_PER_YEAR; // In ray
}
```

Example: Assume an attacker has a P2P supply position on Morpho and wants to earn a very high APY on it. He does the following actions in a single transaction:

- Borrow all funds on the desired Aave market. (This can be done by borrowing against flashloaned collateral).
- The utilisation rate of the market is now 100%. The borrow rate is the max borrow rate and the supply rate is `(1.0 - reserveFactor) * maxBorrowRate`. The max borrow rate can be higher than 100% APY, see Aave docs.
- The attacker triggers an update to the P2P rate, for example, by supplying 1 token to the pool `PositionsManagerForAave.supply(poolTokenAddress, 1, ...)`, triggering `marketsManager.updateSPYs(_poolTokenAddress)`.
- The new mid-rate is computed which will be `(2.0 - reserveFactor) * maxBorrowRate / 2 ~ maxBorrowRate`.
- The attacker repays their Aave debt in the same transaction, not paying any interest on it.
- All P2P borrowers now pay the max borrow rate to the P2P suppliers until the next time a user interacts with the market on Morpho.
- This process can be repeated to keep the APY high.

Recommendation: Consider using a time-weighted mid-rate instead of trusting the current value. Create an oracle contract that is triggered by the protocol administrators to compute the running TWAR (Time-Weighted-Average-Rate) of Aave.

Also, ensure that the P2P rate used by the protocol is updated often (`updateSPYs`) to not diverge from the Aave pool rate. This can be bad in low-activity markets where a high APY is locked in and no P2P update rate is triggered for a long time period.

Morpho: How can this kind of attack be profitable if there is a bot that update rates after such tx? I mean it can only be a griefing attack, right? In the case the user has enough capital to harm Morpho, the user only needs to pay the gas of tx nothing else so it would end up in the same situation no? The drawback of a TWAP would be to unsynch Morpho from the real P2P which can lead other major issues.

Spearbit: You can't assume that there isn't such bot and even if there is, you can't assume that the sync always perfectly ends up in the same block as the attack transaction. Because even if the syncing ends up in the next block you'd still pay high interest for 1 block. This attack only costs gas fees, they can take 0%-fee flashloans (flashmint DAI, turn it to aDAI supply) and then borrow against it. The borrow also doesn't come with any interest because it's repaid in the same block.

It's a griefing attack if the attacker does not have their own supply position on Morpho, but as soon as they have a supply position it's very likely to be profitable because they only pay gas fees but earn high APR.

Morpho's P2P rate is already different from the real Aave mid-rate because it isn't updated every block to reflect changes in Aave's mid-rate. (That's what I mean by lazy-updated snapshot and "Also, ensure that the P2P rate the protocol uses is updated often (`updateSPYs`) to not diverge from the Aave pool rate.")

It doesn't actually have to be time-weighted, it could just be an oracle that also stores the current aave mid-rate but these updates couldn't be manipulated by an attacker as they can only be triggered by admins.

Morpho: We are working on a completely new way to manage exchanges rates. The idea is to delete Morpho's midrate, and update our exchange rates with a formula that only depends on the pool's exchanges rates (and not the rate, to avoid manipulations).

Associated fix in #PR 601.

Spearbit: Acknowledged.

5.2.7 Liquidating Morpho's Aave position leads to state desynchronization

Severity: *High Risk*

Context: PositionsManagerForAaveGettersSetters.sol#L208-L219

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 desynchronize 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: This issue can be prevented by sending, at the beginning at least, aTokens on behalf of Morpho and set it as collateral to prevent this issue. Also we will run our own liquidation bots. We will not implement any "direct" fix inside the code.

Spearbit: Acknowledged, no direct fixes have been implemented.

5.3 Medium Risk

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

Severity: *Medium Risk*

Context: MatchingEngineForAave.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 place 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 sandwich attack mitigations.

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 for 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, matching engine remains unchanged.

5.3.2 TWAP intervals should be flexible as per market conditions

Severity: *Medium Risk*

Context: SwapManagerUniV3.sol#L140-L149

Description: The protocol is using the same `TWAP_INTERVAL` for both weth-morpho and weth-reward token pool while their liquidity and activity might be different. It should use separate appropriate values for both pools.

Recommendation: `TWAP_INTERVAL` value should be changeable (and not constant) by the admin/owner since it is dependent upon market conditions and activity (for e.g 1-hour twap might lag considerably in sudden movements).

Morpho: Valid issue, will fix.

Spearbit: Recommendation has been followed in the PR #557

5.3.3 PositionsManagerForAave claimToTreasury could allow sending underlying to 0x address

Severity: *Medium Risk*

Context: PositionsManagerForAave.sol#L223-L232

Description: `claimToTreasury` is currently not verifying if the `treasuryVault` address is `!= address(0)`. In the current state, it would allow the owner of the contract to burn the underlying token instead of sending it to the intended treasury address.

Recommendation: Add a check to prevent sending treasury underlying tokens to `address(0)` and verify that the `amountToClaim` is `!= 0` to prevent wasting gas and emitting a "false" event.

```
 function claimToTreasury(address _poolTokenAddress)
    external
    onlyOwner
    isMarketCreatedAndNotPaused(_poolTokenAddress)
 {
+   require(treasuryVault != address(0), "treasuryVault != address(0)");
    ERC20 underlyingToken = ERC20(IAToken(_poolTokenAddress).UNDERLYING_ASSET_ADDRESS());
    uint256 amountToClaim = underlyingToken.balanceOf(address(this));
+   require(amountToClaim != 0, "amountToClaim != 0");
    underlyingToken.safeTransfer(treasuryVault, amountToClaim);
    emit ReserveFeeClaimed(_poolTokenAddress, amountToClaim);
 }
```

Morpho: Fixed in PR #562.

Spearbit: Acknowledged.

5.3.4 rewardsManager used in MatchingEngineForAave could be not initialized

Severity: *Medium Risk*

Context: MatchingEngineForAave.sol#L380-L385, MatchingEngineForAave.sol#L410-L415

Description: `MatchingEngineForAave` update the `userUnclaimedRewards` for a supplier/borrower each time it gets updated. `rewardsManager` is not initialized in `PositionsManagerForAaveLogic.initialize` but only via `PositionsManagerForAaveGettersSetters.setRewardsManager`, which means that it will start as `address(0)`.

Each time a supplier or borrower gets updated and the rewardsManager address is empty, the transaction will revert.

To replicate the issue, just comment `positionsManager.setRewardsManager(address(rewardsManager));` in `TestSetup` and run `make c-TestSupply`. All tests will fail with `[FAIL. Reason: Address: low-level delegate call failed]`

Recommendation: Make sure to always call `PositionsManagerForAaveGettersSetters.setRewardsManager` after deploying and initializing `PositionsManagerForAaveLogic` or pass it directly in `PositionsManagerForAaveLogic.initialize` as a new parameter.

Morpho: Check on `address(rewardsManager) != address(0)` has been implemented in PR #554

Spearbit: Acknowledged.

5.3.5 Missing input validation checks on contract initialize/constructor

Severity: *Medium Risk*

Context:

- MarketsManagerForAave.sol#L125-L130
- PositionsManagerForAaveLogic.sol#L24-L42
- RewardsManagerForAave.sol#L63-L66
- SwapManagerUniV2.sol#L54-L70
- SwapManagerUniV3.sol#L58-L85
- SwapManagerUniV3OnEth.sol#L65-L87

Description: Contract initialize/constructor input parameters should always be validated to prevent the creation/initialization of a contract in a wrong/inconsistent state.

Recommendation: Consider implementing the following changes.

`MarketsManagerForAave.sol`

```
 function initialize(ILendingPool _lendingPool) external initializer {
+   require(address(_lendingPool) != address(0), "input != address(0)");

    __UUPSUpgradeable_init();
    __Ownable_init();

    lendingPool = ILendingPool(_lendingPool);
 }
```

`PositionsManagerForAaveLogic.sol`

Important note: `_maxGas` and `NDS` values should also be validated considering the following separated issues:

- Low/high MaxGas values could make match/unmatch supplier/borrower functions always “fail” or revert #34
- NDS min/max value should be properly validated to avoid tx to always fail/skip loop #33

```
 function initialize(
    IMarketsManagerForAave _marketsManager,
    ILendingPoolAddressesProvider _lendingPoolAddressesProvider,
    ISwapManager _swapManager,
    MaxGas memory _maxGas
 ) external initializer {
+   require(address(_marketsManager) != address(0), "_marketsManager != address(0)");
+   require(address(_lendingPoolAddressesProvider) != address(0), "_lendingPoolAddressesProvider !=
↪  address(0)");
+   require(address(_swapManager) != address(0), "_swapManager != address(0)");


    __UUPSUpgradeable_init();
    __ReentrancyGuard_init();
    __Ownable_init();

    maxGas = _maxGas;
    marketsManager = _marketsManager;
    addressesProvider = _lendingPoolAddressesProvider;
    lendingPool = ILendingPool(addressesProvider.getLendingPool());
    matchingEngine = new MatchingEngineForAave();
    swapManager = _swapManager;

    NDS = 20;
 }
```

RewardsManagerForAave.sol

```
 constructor(ILendingPool _lendingPool, IPositionsManagerForAave _positionsManager) {
+   require(address(_lendingPool) != address(0), "_lendingPool != address(0)");
+   require(address(_positionsManager) != address(0), "_positionsManager != address(0)");
    lendingPool = _lendingPool;
    positionsManager = _positionsManager;
 }
```

SwapManagerUniV2.sol

```
 constructor(address _morphoToken, address _rewardToken) {
+   require(_morphoToken != address(0), "_morphoToken != address(0)");
+   require(_rewardToken != address(0), "_rewardToken != address(0)");

    MORPHO = _morphoToken;
    REWARD_TOKEN = _rewardToken;

    /// ...
 }
```

SwapManagerUniV3.sol

Worth noting:

- `_morphoPoolFee` should have a max value check
- `_rewardPoolFee` should have a max value check

```
 constructor(
    address _morphoToken,
    uint24 _morphoPoolFee,
    address _rewardToken,
    uint24 _rewardPoolFee
 ) {
+   require(_morphoToken != address(0), "_morphoToken != address(0)");
+   require(_rewardToken != address(0), "_rewardToken != address(0)");

    MORPHO = _morphoToken;
    MORPHO_POOL_FEE = _morphoPoolFee;
    REWARD_TOKEN = _rewardToken;
    REWARD_POOL_FEE = _rewardPoolFee;

    /// ...
 }
```

`SwapManagerUniV3OnEth.sol`

Worth noting:

- `_morphoPoolFee` should have a max value check

```
 constructor(address _morphoToken, uint24 _morphoPoolFee) {
+   require(_morphoToken != address(0), "_morphoToken != address(0)");

    MORPHO = _morphoToken;
    MORPHO_POOL_FEE = _morphoPoolFee;

    /// ...
 }
```

Morpho: This more our responsibility to set our contracts properly at deployment so we don't think this is much relevant to add these require knowing that the PositionsManager is already a large contract.

However we agree that the 2 following issues are relevant:

- Low/high MaxGas values could make match/unmatch supplier/borrower functions always "fail" or revert #34
- NDS min/max value should be properly validated to avoid tx to always fail/skip loop #33

Spearbit: Acknowledged.

5.3.6 Setting a new rewards manager breaks claiming old rewards

Severity: *Medium Risk*

Context: PositionsManagerForAaveGettersSetters.sol#L62

Description: Setting a new rewards manager will break any old unclaimed rewards as users can only claim through the `PositionManager.claimRewards` function which then uses the new reward manager.

Recommendation: Be cautious when setting new reward managers and ideally ensure there aren't any unclaimed usuer rewards.

Morpho: Perhaps make this setter settable only once? And have another setter saying whether or not we should accrue rewards of users so that in the MatchingEngine we do not call the rewards manager if we already know there is no more liquidity mining.

Spearbit: That is one way to solve it if you don't need the migration behavior.

Morpho: We decided to keep it as it is for now. Will warn users if we plan to change rewards manager. At the end we'll need different reward managers.

Spearbit: Acknowledged, changes have not been made.

5.3.7 Low/high MaxGas values could make match/unmatch supplier/borrower functions always "fail" or revert

Severity: *Medium Risk*

Context: PositionsManagerForAaveGettersSetters.sol#L47-L50, PositionsManagerForAaveLogic.sol#L34

Description: `maxGas` variable is used to determine how much gas the `matchSuppliers`, `unmatchSuppliers`, `matchBorrowers` and `unmatchBorrowers` can consume while trying to match/unmatch supplier/borrower and also updating their position if matched.

- `maxGas = 0` will make entirely skip the loop.
- `maxGas` low would make the loop run at least one time but the smaller maxGas is the higher is the possibility that not all the available suppliers/borrowers are matched/unmatched.
- `maxGas` could make the loop consume all the block gas, making the tx revert.

Note that `maxGas` can be overriden by the user when calling `supply`, `borrow`

Recommendation: Make enough tests to determine a safe min/max value for `maxGas`

Morpho: These parameters will be decided by governance in the future. We will implement a time-lock of seven days to make sure everyone can check the relevance of these parameters. Also the governance has no incentives to implement wrong params that could harm Morpho and its users.

Spearbit: Acknowledged.

5.3.8 NDS min/max value should be properly validated to avoid tx to always fail/skip loop

Severity: *Medium Risk*

Context: PositionsManagerForAaveGettersSetters.sol#L40-L43

Description: `PositionsManagerForAaveLogic` is currently initialized with a default value of `NDS = 20`.

The `NDS` value is used by `MatchingEngineForAave` when it needs to call `DoubleLinkedList.insertSorted` in both `updateBorrowers` and `updateSuppliers`

`updateBorrowers`, `updateSuppliers` are called by

- `MatchingEngineForAavematchBorrowers`
- `MatchingEngineForAaveunmatchBorrowers`
- `MatchingEngineForAavematchSuppliers`
- `MatchingEngineForAaveunmatchSuppliers`

Those functions and also directly `updateBorrowers` and `updateSuppliers` are also called by `PositionsManagerForAaveLogic`

Problems:

- A low `NDS` value would make the loop inside `insertSorted` exit early, increasing the probability of a supplier/borrower to be added to the tail of the list. This is something that Morpho would like to avoid because it would decrease protocol performance when it needs to match/unmatch suppliers/borrowers.
- In the case where a list is long enough, a very high value would make the tranaction revert each time one of those function directly or indirectly call `insertSorted`. The gas "rail guard" present in the match/unmatch supplier/borrow is useless because the loop would be called at least one time.

Recommendation: Make enough tests to determine a safe min/max value for `NDS` that protect from DOS but still make the protocol perform as expected.

Morpho: Fix has been implemented.

Spearbit: Acknowledged.

5.3.9 Initial SwapManager cumulative prices values are wrong

Severity: *Medium Risk*

Context: SwapManagerUniV2.sol#L65-L66

Description: The initial cumulative price values are integer divisions of unscaled reserves and not UQ112x112 fixed-point values.

```
(reserve0, reserve1, blockTimestampLast) = pair.getReserves();
price0CumulativeLast = reserve1 / reserve0;
price1CumulativeLast = reserve0 / reserve1;
```

One of these values will (almost) always be zero due to integer division. Then, when the difference is taken to the real `currentCumulativePrices` in `update`, the TWAP will be a large, wrong value. The slippage checks will not work correctly.

Recommendation: Consider using the same code as the UniswapV2 example oracle.

Morpho: Fixed in PR #550.

Spearbit: Acknowledged.

5.3.10 P2P borrowers' rate can be reduced

Severity: *Medium Risk*

Context: MarketsManagerForAave.sol#L448

Situation: Users on the pool currently earn an inferior rate than users with P2P credit lines. There is a queue for being connected P2P. As this queue cannot br not be fully processed in one single transaction, the protocol introduces the concept of a maximum 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 which leads to worse rates.

i.e: 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 borrow rate) that are still wrongly recorded to be in a P2P position for some borrowers. Such increase in the borrow 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 the better P2P rate than the on-pool rate they earned before.

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

Morpho: We may to refactor the entire queue system at some point.

Spearbit: Acknowledged.

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

Severity: *Medium Risk*

Context: PositionsManagerForAaveLogic.sol#L246

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 Morpho's position on Aave's debt / collateral value is higher than the market's maximum LTV ratio but lower than the market's liquidation threshold, the borrow will fail and the position cannot be liquidated. Therefore 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.

5.4 Low Risk

5.4.1 Event Withdrawn is emitted using the wrong amounts of supplyBalanceInOf

Severity: *Low Risk*

Context: PositionsManagerForAaveLogic.sol#L252-L258

Description: Inside the `_withdraw` function, all changes performed to `supplyBalanceInOf` are done using the `_supplier` address.

The `_receiver` is correctly used only to transfer the `underlying` token via `underlyingToken.safeTransfer(_receiver, _amount);`

The `Withdrawn` event should be emitted passing the `supplyBalanceInOf[_poolTokenAddress]` of the `supplier` and not the receiver.

This problem will arise when this internal function is called by `PositionsManagerForAave.liquidate` where supplier (borrower in this case) and receiver (liquidator) would not be the same address.

Recommendation: Use the `supplier` address to access `supplyBalanceInOf` when emitting the `Withdrawn` event.

```
 emit Withdrawn(
    _supplier,
    _poolTokenAddress,
    _amount,
-    supplyBalanceInOf[_poolTokenAddress][_receiver].onPool,
-    supplyBalanceInOf[_poolTokenAddress][_receiver].inP2P,
+    supplyBalanceInOf[_poolTokenAddress][_supplier].onPool,
+    supplyBalanceInOf[_poolTokenAddress][_supplier].inP2P
 );
```

Morpho: Fixed in the PR #556, event has been moved to the entrypoint contract and uses `msg.sender` as the index which is the supplier.

Spearbit: Acknowledged.

5.4.2 _repayERC20ToPool is approving the wrong amount

Severity: *Low Risk*

Context: PositionsManagerForAaveLogic.sol#L502-L510

Description: `_repayERC20ToPool` is approving the amount of underlying token specified via the input parameter `_amount` when the correct amount that should be approved is the one calculated via:

```
_amount = Math.min(
    _amount,
    variableDebtToken.scaledBalanceOf(address(this)).mulWadByRay(_normalizedVariableDebt)
);
```

Recommendation: Approve the correct amount of underlying token. A possible solution may be as depicted below:

```
-_underlyingToken.safeApprove(address(lendingPool), _amount);
 IVariableDebtToken variableDebtToken = IVariableDebtToken(
    lendingPool.getReserveData(address(_underlyingToken)).variableDebtTokenAddress
 );
// Do not repay more than the contract's debt on Aave
 _amount = Math.min(
    _amount,
    variableDebtToken.scaledBalanceOf(address(this)).mulWadByRay(_normalizedVariableDebt)
 );
+_underlyingToken.safeApprove(address(lendingPool), _amount);
```

Additionally,`variableDebtToken.scaledBalanceOf(address(this)).mulWadByRay(_normalizedVariableDebt)` could be replaced by `variableDebtToken.balanceOf(address(this))` to save gas given how `balanceOf` is implemented on the Aave contract. In this case, the `uint256 \_normalizedVariableDebt` function parameter should be removed.

Morpho: Fixes have been implemented in the PR #536

Spearbit: Acknowledged.

5.4.3 Possible unbounded loop over enteredMarkets array in _getUserHypotheticalBalanceStates

Severity: *Low Risk*

Context: PositionsManagerForAaveLogic.sol#L416

Description: `PositionsManagerForAaveLogic._getUserHypotheticalBalanceStates` is looping `enteredMarkets` which could be an unbounded array leading to a reverted transaction caused by a block gas limit.

While it is true that Morpho will probably handle a subset of assets controlled by Aave, this loop could still revert because of gas limits for a variety of reasons:

- In the future Aave could have more assets and Morpho could match 1:1 those assets.
- Block gas size could decrease.
- Opcodes could cost more gas.

Recommendation: Implement a mechanism that removes `_poolTokenAddress` from the market array to reduce array size if the user does not have more tokens in that specific market.

Morpho: Fixed in PR #560 by possibly exiting the market on withdraw/repay.

Spearbit: Acknowledged.

5.4.4 Wrong liquidation value when withdrawn amount is non-zero

Severity: *Low Risk*

Context: PositionsManagerForAaveLogic.sol#L438-L439

Situation: When `_withdrawnAmount` is non-zero the `liquidationValue` computation uses `assetData.liquidationValue` but should use `assetData.liquidationThreshold` instead.

Recommendation: Currently, this does not lead to any issues as `_withdrawnAmount` is always zero on liquidations and all other calls to `_getUserHypotheticalBalanceStates` ignore this `liquidationValue`. We still recommend fixing this bug in case the return value is used in the future.

```
 liquidationValue -= Math.min(
    liquidationValue,
-   (_withdrawnAmount * assetData.underlyingPrice * assetData.liquidationValue) /
+   (_withdrawnAmount * assetData.underlyingPrice * assetData.liquidationThreshold) /
        (assetData.tokenUnit * MAX_BASIS_POINTS)
 );
```

Morpho: Fixed in the PR #563 according to the recommendation.

Spearbit: Acknowledged.

5.4.5 Missing parameter validation on setters and event spamming prevention

Severity: *Low Risk*

Context:

- RewardsManagerForAave.sol#L72-L79,
- MarketsManagerForAave.sol#L143-L151,
- MarketsManagerForAave.sol#L189-L196,
- MarketsManagerForAave.sol#L200-L202,
- MarketsManagerForAave.sol#L206-L208,
- PositionsManagerForAaveGettersSetters.sol#L33-L36
- PositionsManagerForAaveGettersSetters.sol#L40-L43
- PositionsManagerForAaveGettersSetters.sol#L47-L50
- PositionsManagerForAaveGettersSetters.sol#L54-L57
- PositionsManagerForAaveGettersSetters.sol#L61-L64
- PositionsManagerForAaveGettersSetters.sol#L68-L72

Description: User parameter validity should always be verified to prevent contract updates in an inconsistent state. The parameter's value should also be different from the old one in order to prevent event spamming (emitting an event when not needed) and improve contract monitoring.

`contracts/aave/RewardsManagerForAave.sol`

```
 function setAaveIncentivesController(address _aaveIncentivesController)
    external
    override
    onlyOwner
 {
+    require(_aaveIncentivesController != address(0), "param != address(0)");
+    require(_aaveIncentivesController != aaveIncentivesController, "param != prevValue");
    aaveIncentivesController = IAaveIncentivesController(_aaveIncentivesController);
    emit AaveIncentivesControllerSet(_aaveIncentivesController);
 }
```

contracts/aave/MarketsManagerForAave.sol

```
 function setReserveFactor(address _marketAddress, uint16 _newReserveFactor) external onlyOwner {
-    reserveFactor[_marketAddress] = HALF_MAX_BASIS_POINTS <= _newReserveFactor
-        ? HALF_MAX_BASIS_POINTS
-        : _newReserveFactor;
-
-    updateRates(_marketAddress);
-
-    emit ReserveFactorSet(_marketAddress, reserveFactor[_marketAddress]);
+    require(_marketAddress != address(0), "param != address(0)");
+    uint16 finalReserveFactor = HALF_MAX_BASIS_POINTS <= _newReserveFactor
+        ? HALF_MAX_BASIS_POINTS
+        : _newReserveFactor;
+
+    if( finalReserveFactor !== reserveFactor[_marketAddress] ) {
+        reserveFactor[_marketAddress] = finalReserveFactor;
+        emit ReserveFactorSet(_marketAddress, finalReserveFactor);
+    }
+
+    updateRates(_marketAddress);
 }
```

```
 function setNoP2P(address _marketAddress, bool _noP2P)
    external
    onlyOwner
    isMarketCreated(_marketAddress)
 {
+   require(_noP2P != noP2P[_marketAddress], "param != prevValue");
    noP2P[_marketAddress] = _noP2P;
    emit NoP2PSet(_marketAddress, _noP2P);
 }
```

```
 function updateP2PExchangeRates(address _marketAddress)
    external
    override
    onlyPositionsManager
+   isMarketCreated(_marketAddress)
 {
    _updateP2PExchangeRates(_marketAddress);
 }
```

```
 function updateSPYs(address _marketAddress)
    external
    override
    onlyPositionsManager
+   isMarketCreated(_marketAddress)
 {
    _updateSPYs(_marketAddress);
 }
```

contracts/aave/positions-manager-parts/PositionsManagerForAaveGettersSetters.sol

```
 function setAaveIncentivesController(address _aaveIncentivesController) external onlyOwner {
+   require(_aaveIncentivesController != address(0), "param != address(0)");
+   require(_aaveIncentivesController != aaveIncentivesController, "param != prevValue");
    aaveIncentivesController = IAaveIncentivesController(_aaveIncentivesController);
    emit AaveIncentivesControllerSet(_aaveIncentivesController);
 }
```

Important note: `_newNDS` min/max value should be accurately validated by the team because this will influence the maximum number of cycles that `DDL.insertSorted` can do. Setting a value too high would make the transaction fail while setting it too low would make the `insertSorted` loop exit earlier, resulting in the user being added to the tail of the list.

A more detailed issue about the `NDS` value can be found here: #33

```
 function setNDS(uint8 _newNDS) external onlyOwner {
+   // add a check on `_newNDS` validating correctly max/min value of `_newNDS`
+   require(NDS != _newNDS, "param != prevValue");
    NDS = _newNDS;
    emit NDSSet(_newNDS);
 }
```

Important note: `_newNDS` set to 0 would skip all `theMatchingEngineForAave` match/unmatch supplier/borrower functions if the user does not specify a custom `maxGas`

A more detailed issue about `NDS` value can be found here: #34

```
 function setMaxGas(MaxGas memory _maxGas) external onlyOwner {
+   // add a check on `_maxGas` validating correctly max/min value of `_maxGas`
+   // add a check on `_maxGas` internal value checking that at least one of them is different compared
↪   to the old version
    maxGas = _maxGas;
    emit MaxGasSet(_maxGas);
 }
```

```
 function setTreasuryVault(address _newTreasuryVaultAddress) external onlyOwner {
+   require(_newTreasuryVaultAddress != address(0), "param != address(0)");
+   require(_newTreasuryVaultAddress != treasuryVault, "param != prevValue");
    treasuryVault = _newTreasuryVaultAddress;
    emit TreasuryVaultSet(_newTreasuryVaultAddress);
 }
```

```
 function setRewardsManager(address _rewardsManagerAddress) external onlyOwner {
+   require(_rewardsManagerAddress != address(0), "param != address(0)");
+   require(_rewardsManagerAddress != rewardsManager, "param != prevValue");
    rewardsManager = IRewardsManagerForAave(_rewardsManagerAddress);
    emit RewardsManagerSet(_rewardsManagerAddress);
 }
```

Important note: Should also check that `_poolTokenAddress` is currently handled by the `PositionsManagerForAave` and by the `MarketsManagerForAave`. Without this check a `poolToken` could start in a paused state.

```
 function setPauseStatus(address _poolTokenAddress) external onlyOwner {
+   require(_poolTokenAddress != address(0), "param != address(0)");
    bool newPauseStatus = !paused[_poolTokenAddress];
    paused[_poolTokenAddress] = newPauseStatus;
    emit PauseStatusSet(_poolTokenAddress, newPauseStatus);
 }
```

Recommendation: For each setter, add a validity check on user parameter and a check to prevent to update the state value with the same value and fire an event when it's not needed.

Morpho: After reflection, as all these function will be triggered by governance, It might be overkill to implement all these checks. Although we will implement `min` and `max` value for `NDS` and for `maxGas` values.

Spearbit: Acknowledged.

5.4.6 DDL should prevent inserting items with 0 value

Severity: *Low Risk*

Context: DoubleLinkedList.sol#L83

Description: Currently the DDL library is only checking that the actual value (`_list.accounts[_id].value`) in the list associated with the `_id` is 0 to prevent inserting duplicates.

The DDL library should also verify that the inserted value is greater than 0. This check would prevent adding users with empty values, which may potentially cause the list and as a result the overall protocol to underperform.

Recommendation: Add a require statement to prevent inserting empty values.

```
 function insertSorted(
    List storage _list,
    address _id,
    uint256 _value,
    uint256 _maxIterations
 ) internal {
+   require(_value != 0, "DLL: _value must be != 0");
    require(_list.accounts[_id].value == 0, "DLL: account already created");

    /// other code
 }
```

Note that `require` should be added as soon as possible to also prevent an `SLOAD` from the second `require`.

Morpho: Fixed in PR #526

Spearbit: Acknowledged.

5.4.7 insertSorted iterates more than max iterations parameter

Severity: Low Risk

Context: DoubleLinkedList.sol#L91

Description: The insertSorted function iterates `_maxIterations + 1` times instead of `_maxIterations` times.

Recommendation: Consider changing the code as follows:

Morpho: Fixed in PR #526.

Spearbit: Acknowledged.

5.4.8 insertSorted does not behave like a FIFO for same values

Severity: *Low Risk*

Context: DoubleLinkedList.sol#L93

Description: Users that have the same value are inserted into the list before other users with the same value. It does not respect the "seniority" of the users order and should behave more like a FIFO queue.

Recommendation: Consider introducing the following change:

```
 while (
    numberOfIterations <= _maxIterations &&
    current != _list.tail &&
-   _list.accounts[current].value > _value
+   _list.accounts[current].value >= _value
 )
```

Morpho: Agree, it should behave like in FIFO style in this case, fixed in PR #526.

Spearbit: Acknowledged.

5.4.9 insertSorted inserts elements at wrong index

Severity: *Low Risk*

Context: DoubleLinkedList.sol#L101

Description: The `insertSorted` function inserts elements after the last element has been insterted, when these should have actually been insterted before the last element. The sort order is therefore wrong, even if the maximum iterations count has not been reached. This is because of the check that the `current` element is not the tail.

```
if ( ... && current != _list.tail) { insertBefore } else { insertAtEnd }
```

Example:

- `list = [20]. insert(40)` then `current == list.tail`, and is inserted at the back instead of the front. `result = [20, 40]`
- `list = [30, 10]`, `insert(20)` insertion point should be before `current == 10`, but also `current == tail` therfore the `current != _list.tail` condition is `false` and the element is wrongly inserted at the end. `result = [30, 10, 20]`

Recommendation: Fix the algorithm (while still respecting FIFO, see #25). At some point, the `while` loop breaks and `current` points to some element. If `_list.accounts[current].value < _value`, it means current is a legitimate entry-point to insert before. Otherwise, we insert at the end.

```
function insertSorted(
   List storage _list,
   address _id,
   uint256 _value,
   uint256 _maxIterations
) internal {
   require(_list.accounts[_id].value == 0, "DLL: account already created");

   uint256 numberOfIterations;
   address current = _list.head;
   while (
       numberOfIterations <= _maxIterations &&
       current != _list.tail &&
-      _list.accounts[current].value > _value
+      _list.accounts[current].value >= _value
   ) {
       current = _list.accounts[current].next;
       numberOfIterations++;
   }

   address nextId;
   address prevId;
-    if (numberOfIterations < _maxIterations && current != _list.tail) {
+    if (_list.accounts[current].value < _value) {
       prevId = _list.accounts[current].prev;
       nextId = current;
   } else prevId = _list.tail;

   // ...
}
```

Morpho: Fixed in PR #526.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 PositionsManagerForAaveLogic gas optimization suggestions

Severity: *Gas Optimization*

Context: PositionsManagerForAaveLogic.sol#L91, PositionsManagerForAaveLogic.sol#L145 PositionsManagerForAaveLogic.sol#L201 PositionsManagerForAaveLogic.sol#L305

Description: Update the `remainingTo` variable only when needed. Inside each function, the `remainingTo` counter could be moved inside the `if` statement to avoid calculation when the amount that should be subtracted is `>0`.

Recommendation: Consider implementing the following gas optimization in the `_supply` function:

```
 function _supply(
    address _poolTokenAddress,
    uint256 _amount,
    uint256 _maxGasToConsume
 ) internal isMarketCreatedAndNotPaused(_poolTokenAddress) {
    // ...

    /// Supply in P2P ///

    if (
        borrowersOnPool[_poolTokenAddress].getHead() != address(0) &&
        !marketsManager.noP2P(_poolTokenAddress)
    ) {
        // ...

        if (matched > 0) {
            // ...
+            remainingToSupplyToPool -= matched;
        }
-       remainingToSupplyToPool -= matched;
    }

    /// Supply on pool ///

    // ...
 }
```

Morpho: Implemented as part of the PR #536

Spearbit: Acknowledged.

5.5.2 MarketsManagerForAave._updateSPYs could store calculations in local variables to save gas

Severity: *Gas Optimization*

Context: MarketsManagerForAave.sol#L403-L425

Description: The calculation in the actual code must be updated following this issue: #36. This current issue is an example on how to avoid an additional `SLOAD`.

The function could store locally `currentReserveFactor`, `newSupplyP2PSPY` and `newBorrowP2PSPY` to avoid additional `SLOAD`

Recommendation: Store variable locally to save gas as shown below.

```
uint16 currentReserveFactor = reserveFactor[_marketAddress];
uint256 newSupplyP2PSPY = (meanSPY * (MAX_BASIS_POINTS - currentReserveFactor)) /
    MAX_BASIS_POINTS;
uint256 newBorrowP2PSPY = (meanSPY * (MAX_BASIS_POINTS + currentReserveFactor)) /
    MAX_BASIS_POINTS;

supplyP2PSPY[_marketAddress] = newSupplyP2PSPY;
borrowP2PSPY[_marketAddress] = newBorrowP2PSPY;

emit P2PSPYsUpdated(
    _marketAddress,
    newSupplyP2PSPY,
    newBorrowP2PSPY
);
```

Morpho: Recommendations have been implemented in PR #559

Spearbit: Acknowledged.

5.5.3 Declare variable as immutable/constant and remove unused variables

Severity: *Gas Optimization*

Context:

- RewardsManagerForAave.sol#L29
- RewardsManagerForAave.sol#L30
- RewardsManagerForAave.sol#L31
- SwapManagerUniV2.sol#L27-L28
- SwapManagerUniV2.sol#L33
- SwapManagerUniV3.sol#L33
- SwapManagerUniV3.sol#L35
- SwapManagerUniV3.sol#L41
- SwapManagerUniV3.sol#L42
- SwapManagerUniV3.sol#L43
- SwapManagerUniV3OnEth.sol#L27
- SwapManagerUniV3OnEth.sol#L37
- SwapManagerUniV3OnEth.sol#L38
- SwapManagerUniV3OnEth.sol#L39

Description: Some state variable can be declared as immutable or constant to save gas. Constant variables should be names in uppercase + snake case following the official Solidity style guide. Additionally, variables which are never used across the protocol code can be removed to save gas during deployment and improve readability.

`RewardsManagerForAave.sol`

```
-ILendingPoolAddressesProvider public addressesProvider;

-ILendingPool public lendingPool;
+ILendingPool public immutable lendingPool;

-IPositionsManagerForAave public positionsManager;
+IPositionsManagerForAave public immutable positionsManager;
```

`SwapManagerUniV2.sol`

```
-IUniswapV2Router02 public swapRouter = IUniswapV2Router02(0x60aE616a2155Ee3d9A68541Ba4544862310933d4);
↪  // JoeRouter
+IUniswapV2Router02 public constant SWAP_ROUTER =
↪  IUniswapV2Router02(0x60aE616a2155Ee3d9A68541Ba4544862310933d4); // JoeRouter

-IUniswapV2Pair public pair;
+IUniswapV2Pair public immutable pair;
```

`SwapManagerUniV3.sol`

```
-ISwapRouter public swapRouter = ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564); // The
↪  Uniswap V3 router.
+ISwapRouter public constant SWAP_ROUTER = ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564); //
↪  The Uniswap V3 router.

-address public WETH9; // Intermediate token address.
+address public immutable WETH9; // Intermediate token address.

-IUniswapV3Pool public pool0;
+IUniswapV3Pool public immutable pool0;

-IUniswapV3Pool public pool1;
+IUniswapV3Pool public immutable pool1;

-bool public singlePath;
+bool public boolean singlePath;
```

`SwapManagerUniV3OnEth.sol`

```
-ISwapRouter public swapRouter = ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564); // The
↪  Uniswap V3 router.
+ISwapRouter public constant SWAP_ROUTER = ISwapRouter(0xE592427A0AEce92De3Edee1F18E0157C05861564); //
↪  The Uniswap V3 router.

-IUniswapV3Pool public pool0;
+IUniswapV3Pool public immutable pool0;

-IUniswapV3Pool public pool1;
+IUniswapV3Pool public immutable pool1;

-IUniswapV3Pool public pool2;
+IUniswapV3Pool public immutable pool2;
```

Recommendation:

- Declare selected variable as immutable.
- Declare selected variable as constant and follow the solidity style guide for naming.
- Remove unused variables.

Morpho: Fixes have been implemented in the PR #554. We have also upgraded to Solidity `0.8.13` to leverage the "Reading From Immutable Variables" feature that has been introduced since Solidity `0.8.8`

Spearbit: Acknowledged.

5.5.4 Function does not revert if balance to transfer is zero

Severity: *Gas Optimization*

Context: PositionsManagerForAave.sol#L228-L231

Description: Currently when the `claimToTreasury()` function is called it gets the `amountToClaim` by using `underlyingToken.balanceOf(address(this)`. It then uses this `amountToClaim` in the `safeTransfer()` function and the `ReserveFeeClaimed` event is emitted. The problem is that the function does not take into account that it is possible for the `amountToClaim` to be 0. In this case the `safeTransfer` function would still be called and the `ReserveFeeClaimed` event would still be emitted unnecessarily.

Recommendation: Consider reusing the `AmountIsZero()` custom error in this function to prevent `safeTransfer()` from being called as well as the `ReserveFeeClaimed` event from being emitted if the `amountToClaim` is 0.

```
 ERC20 underlyingToken = ERC20(IAToken(_poolTokenAddress).UNDERLYING_ASSET_ADDRESS());
 uint256 amountToClaim = underlyingToken.balanceOf(address(this));

+ if (amountToClaim == 0) revert AmountIsZero();

 underlyingToken.safeTransfer(treasuryVault, amountToClaim);
 emit ReserveFeeClaimed(_poolTokenAddress, amountToClaim);
```

Morpho: Recommendation added in PR #562

Spearbit: Acknowledged.

5.6 Informational

5.6.1 matchingEngine should be initialized in PositionsManagerForAaveLogic's initialize function

Severity: *Informational*

Context: PositionsManagerForAaveLogic.sol#L38

Description: `MatchingEngineForAave` inherits from `PositionsManagerForAaveStorage` which is an `UUPSUpgradeable` contract.

Following UUPS best practices, the `MatchingEngineForAave` deployed by `PositionsManagerForAaveLogic` should also be initialized.

Recommendation: Initialize `MatchingEngineForAave` when created and deployed by `PositionsManagerForAaveLogic.initialize`

Morpho: Now we are using the transparent proxy pattern instead of the UUPS's one as contract's exceeds max weight for deployment. I propose to ignore this issue or update it in case of the transparent proxy pattern has an issue as well.

Spearbit: Acknowledged, just be sure to follow OZ best practice on contract initialization during the deployment phase.

5.6.2 Misc: notation, style guide, global unit types, etc

Severity: *Informational*

Context: SwapManagerUniV2.sol#L25, SwapManagerUniV3.sol#L28, SwapManagerUniV3.sol#L29, SwapManagerUniV3OnEth.sol#L23, SwapManagerUniV3OnEth.sol#L24, SwapManagerUniV3OnEth.sol#L33, SwapManagerUniV3OnEth.sol#L34, MarketsManagerForAave.sol#L32, MarketsManagerForAave.sol#L33

Description: Follow solidity notation, standard style guide and global unit types to improve readability.

Recommendation: Consider implementing the following changes.

`SwapManagerUniV2.sol`

```
-uint256 public constant MAX_BASIS_POINTS = 10000; // 100% in basis points.
+uint256 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis points.
```

`SwapManagerUniV2.sol`

```
-uint32 public constant TWAP_INTERVAL = 3600; // 1 hour interval.
+uint32 public constant TWAP_INTERVAL = 1 hours; // 1 hour interval.

-uint256 public constant MAX_BASIS_POINTS = 10000; // 100% in basis points.
+uint256 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis points.
```

`SwapManagerUniV3OnEth.sol`

```
-uint32 public constant TWAP_INTERVAL = 3600; // 1 hour interval.
+uint32 public constant TWAP_INTERVAL = 1 hours; // 1 hour interval.

-uint256 public constant MAX_BASIS_POINTS = 10000; // 100% in basis points.
+uint256 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis points.

-uint24 public constant FIRST_POOL_FEE = 3000; // Fee on Uniswap for stkAAVE/AAVE pool.
+uint24 public constant FIRST_POOL_FEE = 3_000; // Fee on Uniswap for stkAAVE/AAVE pool.

-uint24 public constant SECOND_POOL_FEE = 3000; // Fee on Uniswap for AAVE/WETH9 pool.
+uint24 public constant SECOND_POOL_FEE = 3_000; // Fee on Uniswap for AAVE/WETH9 pool.
```

`MarketsManagerForAave.sol`

```
-uint16 public constant MAX_BASIS_POINTS = 10000; // 100% in basis point.
+uint16 public constant MAX_BASIS_POINTS = 10_000; // 100% in basis point.

-uint16 public constant HALF_MAX_BASIS_POINTS = 5000; // 50% in basis point.
+uint16 public constant HALF_MAX_BASIS_POINTS = 5_000; // 50% in basis point.
```

Morpho: Part of the recommendations have been applied inside PR #561

Spearbit: Acknowledged.

5.6.3 Outdated or wrong Natspec documentation

Severity: *Informational*

Context:

- SwapManagerUniV3.sol#L55-L57
- PositionsManagerForAaveGettersSetters.sol#L52
- PositionsManagerForAaveGettersSetters.sol#L115-L120

Description: Some Natspec documentation is missing parameters/return value or is not correctly updated to reflect the function code.

Recommendation: Consider implementing the following changes.

`SwapManagerUniV3.sol`

```
/// @notice Constructs the SwapManagerUniV3 contract.
/// @param _morphoToken The Morpho token address.
+/// @param _morphoPoolFee TODO: ADD CORRECT DOC
/// @param _rewardToken The reward token address.
+/// @param _rewardPoolFee TODO: ADD CORRECT DOC
 constructor(
    address _morphoToken,
    uint24 _morphoPoolFee,
    address _rewardToken,
    uint24 _rewardPoolFee
 ) {
    // ...
 }
```

`PositionsManagerForAaveGettersSetters.sol`

```
-/// @notice Sets the `_newTreasuryVaultAddress`.
+/// @notice Sets the `treasuryVault`.
/// @param _newTreasuryVaultAddress The address of the new `treasuryVault`.
 function setTreasuryVault(address _newTreasuryVaultAddress) external onlyOwner {
    treasuryVault = _newTreasuryVaultAddress;
    emit TreasuryVaultSet(_newTreasuryVaultAddress);
 }
```

```
-/// @notice Returns the collateral value, debt value and max debt value of a given user (in ETH).
+/// @notice Returns the collateral value, debt value, max debt value and liquidation value of a given
↪  user (in ETH).
/// @param _user The user to determine liquidity for.
/// @return collateralValue The collateral value of the user (in ETH).
/// @return debtValue The current debt value of the user (in ETH).
/// @return maxDebtValue The maximum possible debt value of the user (in ETH).
/// @return liquidationValue The value which made liquidation possible (in ETH).
 function getUserBalanceStates(address _user)
    external
    view
    returns (
        uint256 collateralValue,
        uint256 debtValue,
        uint256 maxDebtValue,
        uint256 liquidationValue
    )
 {
    // ...
 }
```

Morpho: Applied in PR #561.

Spearbit: Acknowledged.

5.6.4 Use the official UniswapV3 0.8 branch

Severity: *Informational*

Context: uniswap/*

Description: The current repository creates local copies of the UniswapV3 codebase and manually migrates the contracts to `Solidity 0.8`.

Recommendation: There are issues with the current migration which can be avoided by using the official Uniswap V3 0.8 branch.

Morpho: Agreed, added as dependency in PR #550

Spearbit: Acknowledged.

5.6.5 Unused events and unindexed event parameters

Severity: *Informational*

Context:

- RewardsManagerForAave.sol#L37
- RewardsManagerForAave.sol#L43
- SwapManagerUniV2.sol#L47
- SwapManagerUniV3.sol#L51
- SwapManagerUniV3OnEth.sol#L47
- MarketsManagerForAave.sol#L53
- MarketsManagerForAave.sol#L57
- MarketsManagerForAave.sol#L62
- MarketsManagerForAave.sol#L68-L72
- MarketsManagerForAave.sol#L78-L82
- MarketsManagerForAave.sol#L87
- PositionsManagerForAaveEventsErrors.sol#L97
- PositionsManagerForAaveEventsErrors.sol#L101
- PositionsManagerForAaveEventsErrors.sol#L105
- PositionsManagerForAaveEventsErrors.sol#L110
- PositionsManagerForAaveEventsErrors.sol#L120
- PositionsManagerForAaveEventsErrors.sol#L125
- PositionsManagerForAaveEventsErrors.sol#L131
- PositionsManagerForAaveEventsErrors.sol#L115

Description: Certain parameters should be defined as `indexed` to track them from web3 applications / security monitoring tools.

Recommendation: Define such parameters as `indexed` and remove unused event from the code base.

`contracts/aave/RewardsManagerForAave.sol`

```
- event AaveIncentivesControllerSet(address _aaveIncentivesController);
+ event AaveIncentivesControllerSet(address indexed _aaveIncentivesController);

- event UserIndexUpdated(address _user, address _poolTokenAddress, uint256 _index);
+ event UserIndexUpdated(address indexed _user, address indexed _poolTokenAddress, uint256 _index);
```

`contracts/common/SwapManagerUniV2.sol`

```
- event Swapped(address _receiver, uint256 _amountIn, uint256 _amountOut);
+ event Swapped(address indexed _receiver, uint256 _amountIn, uint256 _amountOut);
```

`contracts/common/SwapManagerUniV3.sol`

```
- event Swapped(address _receiver, uint256 _amountIn, uint256 _amountOut);
+ event Swapped(address indexed _receiver, uint256 _amountIn, uint256 _amountOut);
```

`contracts/common/SwapManagerUniV3.sol`

```
- event Swapped(address _receiver, uint256 _amountIn, uint256 _amountOut);
+ event Swapped(address indexed _receiver, uint256 _amountIn, uint256 _amountOut);
```

contracts/aave/MarketsManagerForAave.sol

```
- event MarketCreated(address _marketAddress);
+ event MarketCreated(address indexed _marketAddress);

- event PositionsManagerSet(address _positionsManager);
+ event PositionsManagerSet(address indexed _positionsManager);

- event NoP2PSet(address _marketAddress, bool _noP2P);
+ event NoP2PSet(address indexed _marketAddress, bool _noP2P);

-event P2PSPYsUpdated(
-    address _marketAddress,
-    uint256 _newSupplyP2PSPY,
-    uint256 _newBorrowP2PSPY
-);
+event P2PSPYsUpdated(
+    address indexed _marketAddress,
+    uint256 _newSupplyP2PSPY,
+    uint256 _newBorrowP2PSPY
+);

- event P2PExchangeRatesUpdated(
-    address _marketAddress,
-    uint256 _newSupplyP2PExchangeRate,
-    uint256 _newBorrowP2PExchangeRate
-);
+ event P2PExchangeRatesUpdated(
+    address _marketAddress,
+    uint256 _newSupplyP2PExchangeRate,
+    uint256 _newBorrowP2PExchangeRate
+);
```

contracts/aave/positions-manager-parts/PositionsManagerForAaveEventsErrors.sol

```
-event TreasuryVaultSet(address _newTreasuryVaultAddress);
+event TreasuryVaultSet(address indexed _newTreasuryVaultAddress);

-event RewardsManagerSet(address _newRewardsManagerAddress);
+event RewardsManagerSet(address indexed _newRewardsManagerAddress);

-event AaveIncentivesControllerSet(address _aaveIncentivesController);
+event AaveIncentivesControllerSet(address indexed _aaveIncentivesController);

-event PauseStatusSet(address _poolTokenAddress, bool _newStatus);
+event PauseStatusSet(address indexed _poolTokenAddress, bool _newStatus);

-event ReserveFeeClaimed(address _poolTokenAddress, uint256 _amountClaimed);
+event ReserveFeeClaimed(address indexed _poolTokenAddress, uint256 _amountClaimed);

-event RewardsClaimed(address _user, uint256 _amountClaimed);
+event RewardsClaimed(address indexed _user, uint256 _amountClaimed);

-event RewardsClaimedAndSwapped(address _user, uint256 _amountIn, uint256 _amountOut);
+event RewardsClaimedAndSwapped(address indexed _user, uint256 _amountIn, uint256 _amountOut);
```

The following events should be removed from code base because they are never used

contracts/aave/positions-manager-parts/PositionsManagerForAaveEventsErrors.sol

```
-event FeesClaimed(address _poolTokenAddress, uint256 _amountClaimed);
```

Morpho: Fixes has been implemented in the PR #552

Spearbit: Acknowledged.

5.6.6 Rewards are ignored in the on-pool rate computation

Severity: *Informational*

Context: Protocol level

Description: Morpho claims that the protocol is a strict improvement upon the underlying lending protocols. It tries to match as many suppliers and borrowers P2P at the supply/borrow mid-rate of the underlying protocol. However, given high reward incentives paid out to on-pool users it could be the case that being on the pool yields a better rate than the P2P rate.

Recommendation: While Morpho pays forward the reward incentives to on-pool users, they try to match all users P2P first. Users should be aware of this fact and lend/borrow directly from the underlying protocol instead of being matched P2P by Morpho if they want to maximize their rates.

Morpho: This is the purpose of noP2P which disables P2P matching of a given market.

Spearbit: Acknowledged.

5.6.7 User transfer AToken to Morpho or deposit on behalf of Morpho

Severity: *Informational*

Context: Protocol Level

Description: Link to Aave LendinPool.sol

> onBehalfOf The address that will receive the aTokens, same as msg.sender if the user wants to receive them on his own wallet, or a different address if the beneficiary of aTokens is a different wallet

This method could allow anyone to deposit some underlying on Aave, mint the respective AToken that will be transferred to Morpho (address(morphoPositionManager)).

The result is the same as if someone would directly transfer to Morpho (`address(morphoPositionManager)`) some AToken.

Consequences:

- Morpho would have "stuck" free AToken on `address(positionManager)` that cannot be redeemed for underlying.
- Morpho's health factor on Aave would be better because it's like it's supplying more collateral.

Morpho: No real reason to fix this. It is true for all contracts.

Spearbit: Acknowledged.