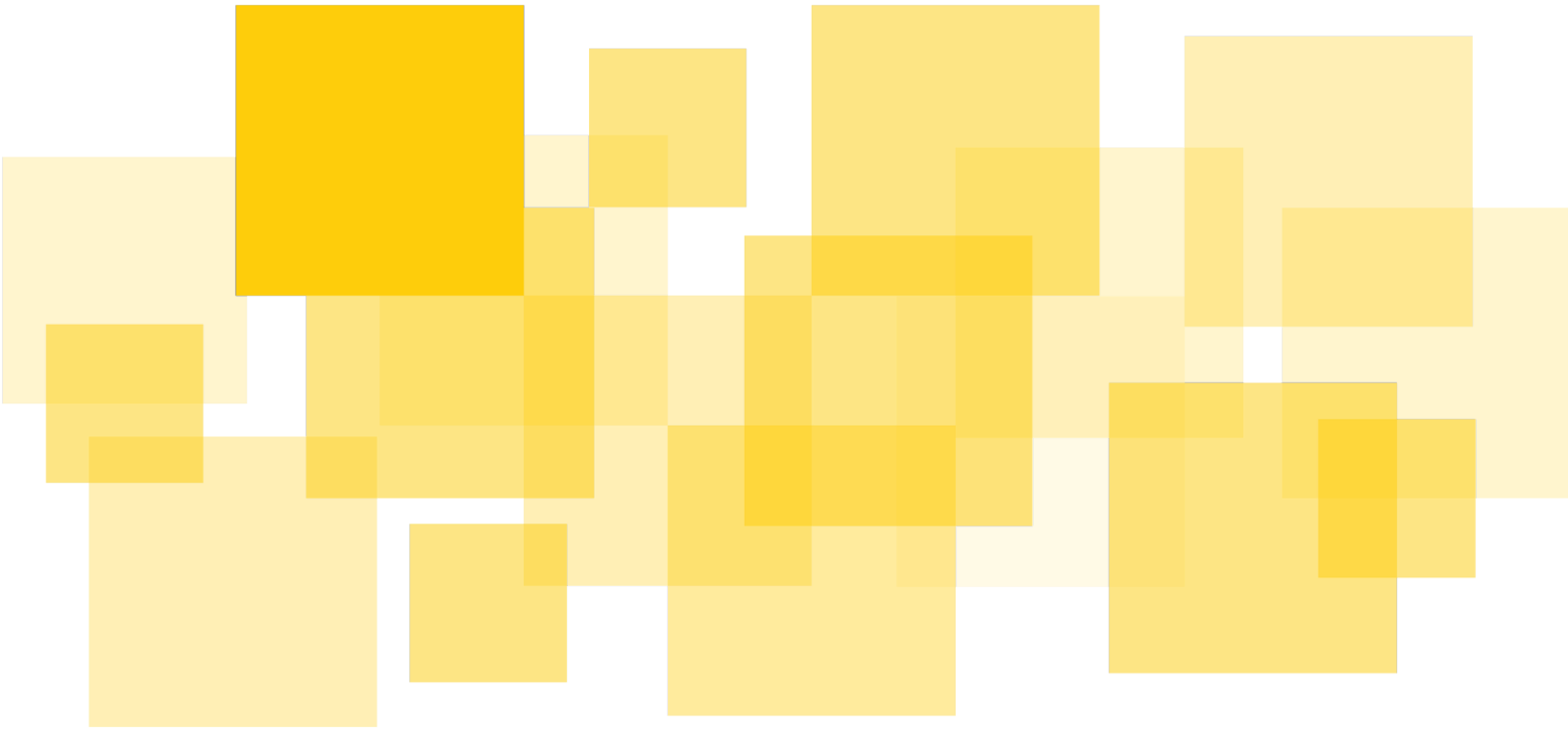
Audit Report

Morpho

Delivered: May 2, 2023



Prepared for Morpho Labs by Runtime Verification, Inc.

runtime verification

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carry substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Summary

Runtime Verification, Inc. has audited the smart contract source code of Morpho AAVE v3. The review was conducted from 2023-01-30 to 2023-03-24. A follow-up review focused on bug fixes was conducted from 2023-04-17 to 2023-04-28.

Morpho Labs engaged Runtime Verification in checking the security of their Morpho AAVE v3 protocol.

The issues which have been identified can be found in the section General Findings.

Scope

The audited smart contracts are:

- `MatchingEngine.sol`
- `Morpho.sol`
- `MorphoGetters.sol`
- `MorphoSetter.sol`
- `MorphoInternal.sol`
- `MorphoStorage.sol`
- `PositionsManager.sol`
- `PositionsManagerInternal.sol`
- `RewardsManager.sol`
- `libraries/Constants.sol`
- `libraries/DeltasLib.sol`
- `libraries/Errors.sol`
- `libraries/Events.sol`
- `libraries/InterestRatesLib.sol`
- `libraries/MarketBalanceLib.sol`
- `libraries/MarketLib.sol`
- `libraries/MarketSideDeltaLib.sol`
- `libraries/PoolLib.sol`
- `libraries/ReserveDataLib.sol`
- `libraries/Types.sol`
- `extensions/WETHGateway.sol`

The audit has focused on the above contracts and has assumed the correctness of the libraries and external contracts they use. The libraries are widely used and assumed to be secure and functionally correct.

The review focused mainly on the `morpho-dao/morpho-aave-v3` private code repository. The code was frozen for review at commit `76525db269`.

The follow-up audit of the fixes took the differences into account that were introduced by the following merge-commits:

- https://github.com/morpho-dao/morpho-aave-v3/commit/178ae65df5482414fda0af91c6e0d0cc0b0c911e
- https://github.com/morpho-dao/morpho-aave-v3/commit/51afdd50da9b32b2dddd58ec1c32f281474d9dca
- https://github.com/morpho-dao/morpho-aave-v3/commit/37271974b0a7df139256dd3d4b5bd72dbc58d409
- https://github.com/morpho-dao/morpho-aave-v3/commit/252082ac23674ff818b9b03ab934e5b12022e204
- https://github.com/morpho-dao/morpho-aave-v3/commit/f42d7d7d35582565a8a98919bcafd5851f8f54ae
- https://github.com/morpho-dao/morpho-aave-v3/commit/3f33f2f4571b8244b4f01b2fad9b1f8e3ceaad4f
- https://github.com/morpho-dao/morpho-aave-v3/commit/7781520d8b578a2be37d3fd5c521f3a0cc405db3
- https://github.com/morpho-dao/morpho-aave-v3/commit/1cedc9ce44cd20f1721df5fa06be2704bd814092
- https://github.com/morpho-dao/morpho-aave-v3/commit/e8818e36a9fd5ab950d63c142d2b88b13924ba76
- https://github.com/morpho-dao/morpho-aave-v3/commit/d79974a0ab0d1dbf4053269cd31cf4bc9a310e37
- https://github.com/morpho-dao/morpho-aave-v3/commit/5c1df9978720a4be9fe03bcfcb46f3571e3e4034
- https://github.com/morpho-dao/morpho-aave-v3/commit/d697acdc92a55f4a3d9d0abf90e2e582c2a3f1e1
- https://github.com/morpho-dao/morpho-aave-v3/commit/7f8fd4f574960e874ad6b6a7e8e32049dfc3cc0b
- https://github.com/morpho-dao/morpho-aave-v3/commit/47a485a31899e420a11a6cc748624f3e54e389e7
- https://github.com/morpho-dao/morpho-aave-v3/commit/3214e76cb1aa21093d955cef24955476bd7f604b
- https://github.com/morpho-dao/morpho-aave-v3/commit/c4f0f6412408442407db2296e5c1551528da3cfc
- https://github.com/morpho-dao/morpho-aave-v3/commit/4d510b8ba69b8ee411d205fb8b425aa5f9676cfb
- https://github.com/morpho-dao/morpho-aave-v3/commit/a4d9a14a37234fbd682d374964aa1743dbbcf795
- https://github.com/morpho-dao/morpho-aave-v3/commit/450b91ffb08c53ea2f7910cf22175119d236ee76
- https://github.com/morpho-dao/morpho-aave-v3/commit/aea72b1419286aa02678b13239a460eaee5187fc
- https://github.com/morpho-dao/morpho-aave-v3/commit/178ae65df5482414fda0af91c6e0d0cc0b0c911e
- https://github.com/morpho-dao/morpho-aave-v3/commit/51afdd50da9b32b2dddd58ec1c32f281474d9dca
- https://github.com/morpho-dao/morpho-aave-v3/commit/0cc17dfe48fa5ed9d37b5da735c8759921e5e630
- https://github.com/morpho-dao/morpho-aave-v3/commit/e08e5f620bf1059530b9af018b94d21973e8287c
- https://github.com/morpho-dao/morpho-aave-v3/commit/d30221201fa58a89deec9ca9194aca1cfdf5f0a2
- https://github.com/morpho-dao/morpho-aave-v3/commit/636f0593732aa98f6574eb7eef6668acda83edc0
- https://github.com/morpho-dao/morpho-aave-v3/commit/2d1c0007be8d18c76053ea83df1fc034f955fce3
- https://github.com/morpho-dao/morpho-aave-v3/commit/b00cf80a26bed1fb955ba3d38b8b7c14dd43413f
- https://github.com/morpho-dao/morpho-aave-v3/commit/40694aba8d46e0f6640676792258f999561e3df7
- https://github.com/morpho-dao/morpho-aave-v3/commit/b16b412e045ec04f662a756ab80c9b64385d3255
- https://github.com/morpho-dao/morpho-aave-v3/commit/b4c35140346e85c24ea91f044b442d82c95cd5e2
- https://github.com/morpho-dao/morpho-aave-v3/commit/22070bcefe644e0f1f2c3d2a7af7a68cf500fc54
- https://github.com/morpho-dao/morpho-aave-v3/commit/86713a2cd243ae262c1d524447beee939cb8bf64
- https://github.com/morpho-dao/morpho-aave-v3/commit/26ca169e232e34472378f5ec1b5137172f58a8f2
- https://github.com/morpho-labs/morpho-data-structures-private/commit/e0e4745d4312b55ee405f372a489abf753b376fe
- https://github.com/morpho-labs/morpho-data-structures-private/commit/fd2027a4d3e8908434e1fd35cbbb815a5f1a8ff6

Assumptions:

The audit assumes that all addresses assigned a role must be trusted for as long as they hold that role. The section trust model lists the detailed capabilities and responsibilities of each role.

Note the assumption assumes honesty and competence. However, we will rely less on competence and point out how the contracts could better ensure unintended mistakes cannot happen wherever possible.

Methodology

Although the manual code review cannot guarantee to find all potential security vulnerabilities as mentioned in the Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and inconsistency between the logic and the implementation. Second, we carefully checked if the code was vulnerable to known security issues and attack vectors. We applied various code quality assurance tools to aid our analysis, ranging from static analysis tools, e.g. Slither, over dynamic testing tools, e.g. Foundry, to symbolic execution with KEVM. Finally, we met with the Morpho team to provide feedback and suggested development practices.

This report describes the **intended** behavior and invariants of the contracts under review. Then it outlines issues we have found, both in the intended behavior and how the code differs from it. We also point out lesser concerns, deviations from best practices, and any other weaknesses we encounter. Finally, we also give an overview of the critical security properties we proved during the review.

Protocol Model

Trust Model

Each Morpho instance has an owner with elevated privileges. These privileges can be used to (re-)configure the instance, update risk parameters, or to pause certain features. The deployer of the Morpho instance will be the first owner. The ownership can be transferred in a two-step process to any other address.

The Morpho-owner account has permissions to execute the following functions:

- `transferOwnership`
- `createMarket`
- `claimToTreasury`
- `increaseP2PDeltas`
- `setDefaultIterations`
- `setPositionsManager`
- `setRewardsManager`
- `setTreasuryVault`
- `setReserveFactor`
- `setP2PIndexCursor`
- `setIsSupplyPaused`
- `setIsSupplyCollateralPaused`
- `setIsBorrowPaused`
- `setIsRepayPaused`
- `setIsWithdrawPaused`
- `setIsWithdrawCollateralPaused`
- `setIsLiquidateCollateralPaused`
- `setIsLiquidateBorrowPaused`
- `setIsPaused`
- `setIsPausedForAllMarkets`
- `setIsP2PDisabled`
- `setIsDeprecated`

Notice, that Morpho Labs plans to deploy the main contract behind an upgradable proxy. Consequently, the operator of the proxy contract has the capability to replace the implementation contract at any time.

Further notice, that a Morpho instance is connected to an underlying AAVE pool. AAVE pools expose some privileged functions with elevated capabilities. For example, the AAVE EMERGENCY_ADMIN can pause a pool. In this case, Morpho is not fully operational.

The interested reader is advised to see the following link for an overview of AAVEs trust model. https://docs.aave.com/developers/core-contracts/aclmanager

Contract Invariants

This section identifies some critical assumptions and contract invariants. Notice, that some invariants are simplifications and need be refined to match the exact semantics of Morpho. Please refer to the Glossary to understand the intention of the various symbols.

For some of the invariants we have developed automated fuzz tests that can check the contracts conformance. Notice, that not all variables are directly observable on the Solidity level. For example, the total underlying balances S^{P2P}, S^{Pool}, B^{P2P}, B^{Pool} are not explicitly maintained in the contract storage. As a consequence, they were introduced as ghost variables in our tests.

General Assumptions

The following assumptions should be made for the following invariants to hold.

Assumption 1: Pool rates boundaries

The underlying pool supply rate is not smaller than the underlying pool borrow rate: $r^S \geq r^B$

Assumption 2: Compound interest on underlying pools

The protocol assumes that Aave is using the compound interest mechanism.

Assumption 3: No fee-taking tokens

We assume that no integrated ERC-20 charges fees during transfers.

Internal Balances

I1: Total P2P supply

The total supply in P2P positions is equal to the sum of the supplies in P2P positions.

$$S^{P2P} = \sum_{a \in A} s^{P2P}(a)$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

I2: Total pool supply

The total supply in pool position is equal to the sum of the supplies in Pool positions.

$$S^{Pool} = \sum_{a \in A} s^{Pool}(a)$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

I3: Total P2P borrow amount

The total borrowed amount in P2P positions is equal to the sum of all borrowed amounts in P2P positions.

$$B^{P2P} = \sum_{a \in A} b^{P2P}(a)$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

I4: Total pool borrow amount

The total borrowed amount in pool position is equal to the sum of all borrowed amounts in pool positions.

$$B^{Pool} = \sum_{a \in A} b^{Pool}(a)$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

I5: Equal amounts in P2P positions

The sum of all P2P-supplies is equal to the sum of all P2P-borrows. Notice, that this only holds in the absence of deltas and a reserve factor of exactly 1.

$$S^{P2P} = B^{P2P}$$

$$\sum_{a \in A} s^{P2P}(a) = \sum_{a \in A} b^{P2P}(a)$$

If we take deltas into account, we can refine the invariant like follows:

$$S^{P2P} - \delta^{S} = B^{P2P} - \delta^{B}$$

$$\sum_{a \in A} s^{P2P}(a) - \delta^{S} = \sum_{a \in A} b^{P2P}(a) - \delta^{B}$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

User positions, idle supply, collateral supply

I6: A user's total supply is matched P2P or on-pool

The total balance of a user is either matched in a P2P position or in the pool supply position.

$$S(a) = s^{P2P}(a) + s^{Pool}(a)$$

$S(a)$ is a ghost variable.

I7: The total supply is matched P2P or on-pool

The total balance of all users is either matched in a P2P position or in the pool supply position.

$$S = \sum_{a \in A} S(a)$$

S is a ghost variable.

I8: A users total borrowed amount is matched P2P or on-pool

The total borrowed amount of a user is either matched in a P2P position or in the pool borrow position.

$$B(a) = b^{P2P}(a) + b^{Pool}(a)$$

$B(a)$ is a ghost variable.

I9: The total borrowed amount is matched P2P or on-pool.

The total borrowed amount of all users is either matched in a P2P position or in the pool supply position.

$$B = \sum_{a \in A} B(a)$$

B is a ghost variable.

AAVE Integration

I10: Morpho's supply position on Aave

Morpho's supply position on AAVE is equal to the total supply in pool positions. Notice, that this only holds in the absence of deltas and pure collateral supply.

$$s(M) = S^{Pool}$$

After taking deltas into account, we can refine the invariant like follows.

$$s(M) = S^{Pool} + \delta^{S}$$

Further taking the pure collateral supply into account the invariant becomes:

$$s(M) = S^{Pool} + \delta^{S} + C$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

I11: Morpho's borrow position on Aave

Morpho's borrow position on AAVE is equal to the total borrowed amount in pool positions. Notice, that this only holds in the absence of deltas.

$$b(M) = B^{Pool}$$

After taking deltas into account, we can refine the invariant like follows.

$$b(M) = B^{Pool} + \delta^{B}$$

As part of our analysis we developed a Foundry invariant test that covers this contract invariant.

Interest Rates Mechanism

This section presents functional correctness properties of P2P index calculation for the general case and for the cases where calculation is performed for borrow and supply operations.

Following table summarizes the symbols used in properties

Symbol	Description
$\dot{\lambda}^{Pool}$	Pool growth factor
$\dot{\lambda}^{P2P}$	P2P growth factor
$\lambda_{t'}^{Pool}$	Last pool index
$\lambda_{t'}^{P2P}$	Last P2P index
$\lambda_{t'}^{\gamma^{\Sigma}} \in \{\lambda_{t'}^{\gamma^{S}}, \lambda_{t'}^{\gamma^{B}}\}$	Last P2P supply/borrow index (respecting deltas, reserve factor and P2P index cursor)
$\delta^{\Sigma} \in \{\delta^{S}, \delta^{B}\}$	Scaled delta value
$\Sigma^{P2P} \in \{B^{P2P}, S^{P2P}\}$	Scaled P2P total value
$\gamma^{idle} \in [0, 1]$	Proportion of idle supply to total P2P supply
$\gamma^{\Sigma} \in [0, 1]$	Supply/borrow share

General Case:

Pre-condition: $\lambda_{t'}^{\gamma^{\Sigma}} \geq 1 \wedge \gamma^{idle} \in [0, 1] \wedge \gamma^{\Sigma} \in [0, 1]$

Command: $\lambda_{t}^{\gamma^{\Sigma}} = computeP2PIndex(\dot{\lambda}^{Pool}, \dot{\lambda}^{P2P}, \lambda_{t'}^{Pool}, \lambda_{t'}^{P2P}, \delta^{\Sigma}, \Sigma^{P2P}, \gamma^{idle})$

Post-condition:

$$\lambda_{t}^{\gamma^{\Sigma}} = \begin{cases} \lambda_{t'}^{\gamma^{\Sigma}} \dot{\lambda}^{P2P} & \text{if } \Sigma^{P2P} = 0 \ \vee (\delta^{\Sigma} = 0 \ \wedge \gamma^{idle} = 0) \\ \lambda_{t'}^{\gamma^{\Sigma}} (\dot{\lambda}^{P2P}(1 - \gamma^{\Sigma} - \gamma^{idle}) + \dot{\lambda}^{Pool}\gamma^{\Sigma} + \gamma^{idle}) & \text{otherwise} \end{cases}$$

where

$$\gamma^{\Sigma} = \lceil \frac{\delta^{\Sigma} \lambda_{t}^{Pool}}{\Sigma^{P2P} \lambda_{t}^{P2P}} \rceil$$

Borrow Case:

Pre-condition: $\lambda_{t'}^{\gamma^S} \geq 1 \wedge \gamma^B \in [0,1]$

Command: $\lambda_t^{\gamma^B} = computeP2PIndex(\dot{\lambda}^B, \dot{\lambda}^{\gamma^B}, \lambda_{t'}^B, \lambda_{t'}^{\gamma^B}, \delta^B, B^{P2P}, 0)$

Post-condition:

$$\lambda_t^{\gamma^B} = \begin{cases} \lambda_{t'}^{\gamma^B} \dot{\lambda}^{\gamma^B} & \text{if } B^{P2P} = 0 \ \vee \delta^B = 0 \\ \lambda_{t'}^{\gamma^B} (\dot{\lambda}^{\gamma^B}(1-\gamma^B) + \dot{\lambda}^B \gamma^B) & \text{otherwise} \end{cases}$$

where

$$\gamma^B = \lceil \frac{\delta^B \lambda^B}{B^{P2P} \lambda_{t'}^{\gamma^B}} \rceil$$

Supply Case:

Pre-condition: $\lambda_{t'}^{\gamma^S} \geq 1 \wedge \gamma^{idle} \in [0,1] \wedge \gamma^S \in [0,1]$

Command: $\lambda_t^{\gamma^B} = computeP2PIndex(\dot{\lambda}^S, \dot{\lambda}^{\gamma^S}, \lambda_{t'}^S, \lambda_{t'}^{\gamma^S}, \delta^S, S^{P2P}, \gamma^{idle})$

Post-condition:

$$\lambda_t^{\gamma^S} = \begin{cases} \lambda_{t'}^{\gamma^S} \dot{\lambda}^{\gamma^S} & \text{if } S^{P2P} = 0 \ \vee (\delta^S = 0 \ \wedge \gamma^{idle} = 0) \\ \lambda_{t'}^{\gamma^S} (\dot{\lambda}^{\gamma^S}(1-\gamma^S-\gamma^{idle}) + \dot{\lambda}^S \gamma^S + \gamma^{idle}) & \text{otherwise} \end{cases}$$

where

$$\gamma^S = \lceil \frac{\delta^S \lambda^S}{S^{P2P} \lambda_{t'}^{\gamma^S}} \rceil$$

Testing and mechanized formal verification

We transformed the above math equations into a Solidity specification. This specification is directly executable with Foundry's forge and Runtime Verification's KEVM. Forge is a testing framework where tests can be parametric. When Forge executes the test it will automatically choose random inputs for all its parameters. The benefit of a fuzz test over a traditional unit test is that it covers a larger portion of the input domain. Forge is also really fast, and it can often

quickly find counter-examples. When Forge succeeds it means it did not find a counter-example. This is where KEVM steps in. KEVM reuses the same specification format but instead of choosing a random inputs for the test parameters, KEVM leaves the parameter fully symbolic and tries to test 100% of the input domain at once. This technique is called symbolic execution. If KEVM does not find a counter-example and successfully terminates, it means that there is no counter-example, or to phrase it positively: We have established a proof that the desired property always holds.

The functional correctness specification of the interest rates mechanism was provided to Morpho Labs as an additional artifact of this audit process. Forge successfully tested the implementation against this specification, and KEVM successfully proved the conformance.

General Findings

A01: Rounding errors in PositionManagerInternal.sol

[Severity: Mid | Difficulty: Low | Category: Protocol Invariants]

The PositionManagerInternal contract is the main entry point for the internal bookkeeping of user balances. Since Solidity has no native support for arbitrary precision arithmetic, the accounting depends on approximations of the actual underlying quantities. The rounding direction (e.g. flooring, ceiling, rounding half up) is critical to the safety of the user funds.

Suppose an approximation is performed from the wrong direction, such as underestimated, where it should be overestimated. In that case, it opens up the attack surface to drain assets from the contract. See the scenario for detailed instructions.

The following functions suffer from rounding direction mistakes.

- ☑ A checkmark means the rounding direction is correct.
- ☐ A missing checkmark means the rounding direction is incorrect.

_accountSupply

The user is sending assets to the protocol. The contract updates the internal scaled supply balances of the user. Hence, the user's position should be underestimated.

- ☑ onPool uses flooring division.
- ☐ inP2P is rounded half up. Should use flooring division.

_accountBorrow

The user is taking assets out of the protocol. The contract updates the internal scaled borrow balances of the user. Hence, the user's position should be overestimated.

- ☐ onPool uses flooring division. Should use ceiling division.
- ☐ inP2P is rounded half up. Should use ceiling division.

_accountRepay

The user is sending assets to the pool. The contract updates the scaled borrow balances of the user. Hence, the user's position should be overestimated.

- ☐ onPool is underestimated. It should be overestimated.

- [ ] inP2P is underestimated. It should be overestimated.

_accountWithdraw

The user is taking assets out of the protocol. The contract updates the internal scaled supply balances of the user. Hence, the user's position should be underestimated.

- [ ] onPool is overestimated. It should be underestimated.
- [ ] inP2P is overestimated. It should be underestimated.

_accountSupplyCollateral

The user is sending assets to the protocol. The contract updates the internal scaled collateral balances of the user. Hence, the user's position should be underestimated.

- [x] collateralBalance should be underestimated.

_accountWithdrawCollateral

The user is taking assets out of the protocol. The contract updates the internal scaled collateral balances of the user. Hence, the user's position should be underestimated.

- [ ] collateralBalance should be underestimated.

Scenario

The following example demonstrates the issue in a simplified setting. Here the supply function is over-estimating a value that should be underestimated.

1. Assume that the pool index for some fixed asset is 2.
2. Assume that the sum of borrows in P2P positions is 0. This is only for the simplicity of the example and does not cause a loss of generality.
3. Alice supplies 1 token. We must divide the supplied amount (1) by the index (2) to update the internal scaled supply balance.
4. The exact result would be 1/2. However, due to rounding, the stored balance is 1.
5. Alice can now withdraw 2 tokens. We first divide the withdrawn amount (2) by the index (2), which gives us 1 scaled token.
6. We have to subtract this value (1) from Alice's internal balance (1). Notice that this does not lead to an underflow.

The example demonstrates how Alice can drain assets from the contract by calling supply/withdraw.

Classification

The issue is classified as medium severity because it can lead to a loss of small amounts of user funds. We stress that the severity depends on the ERC20 implementation of the underlying reserve and the economical value that the token carries. In particular, the issue becomes more pressing when the the underlying reserve has a higher price per WAD. Also, tokens with lower decimal precisions are at a higher risk. The issue is classified as low difficulty, because any user can predict and exploit the scenario.

Recommendation

To avoid this scenario, it is recommended to perform all rounding operations so that the dust is kept inside the contract and not sent out to the users. For example, if the contract had used flooring division in step 4, Alice's attempt to withdraw 2 tokens in step 5 would have reverted because the withdrawal would have exceeded her balance.

A02: AAVE pool address changes are not handled

[Severity: Mid | Difficulty: High | Category: -]

AAVE pools are upgradable and the address of a liquidity pool can change after an update. Hence, AAVE recommends using the so-called address provider to obtain the address of a pool.

> *Whenever the LendingPool contract is needed, we recommended you fetch the correct address from the LendingPoolAddressesProvider smart contract, show in the example below.*
>
> https://docs.aave.com/developers/v/1.0/developing-on-aave/the-protocol/lendingpooladdressesprovider

However, Morpho only obtains the address of the pool via the address provider once during deployment. The address is then cached for the lifetime of the contract. If the address of the underlying AAVE pool is changed later, the change is not reflected on Morpho. In this case the Morpho instance is connected to an outdated AAVE liquidity pool. On the other hand, we point out that there is no graceful way to handle upgrades of AAVE pools purely on-chain. Upgrades may come for different reasons, and introduce bug-fixes or new features.

Classification

The issue is classified as medium severity, because the system security relies on assumptions about externalities. It's classified as high difficulty because only privileged AAVE users can upgrade liquidity pools.

Recommendation

Morpho Lab should inform it's users prominently and transparently about planned and unplanned pool updates as soon as possible. Further, each change of the upgraded contract should be reviewed with a focus on the implications that this change has on the Morpho protocol.

Status

Morpho Labs acknowledges the issue.

A03: Risk of infinite approvals

[Severity: High | Difficulty: High | Category: Security]

When creating a market, Morpho grants infinite approval to AAVE for the underlying reserve. There are several risks related to infinite approvals:

1. If an attacker manages to gain control over AAVE, the attacker might also be able to drain assets from Morpho. In particular, this affects the idle supplies sitting on Morpho.
2. Not all tokens implement infinite approvals. Some tokens will revert if the approved amount exceeds the user balance or some pre-configured threshold. In this case, the createMarket call will also revert. Hence, creating markets for such tokens on Morpho is impossible.
3. Some tokens will decrement the approved amount on every transferFrom call, even when the approval was set to type(uint256).max. In this case, a malicious user can repeatedly decrement the allowances granted from Morpho to AAVE of a specific reserve until it reaches a critically low value. When this happens, AAVE will no longer be able to pull in reserves from Morpho. Consequently, supply and repayment actions to AAVE can fail, which can cause supply, repayment, and liquidate activities on Morpho to fail.

We performed a best effort analysis of the following underlying reserve tokens to rule out the possibility of the second and third issue described above. Notice, that these tokens are not in the scope of the audit and the analysis was not performed with the same rigor as for the contracts in scope. To this end, we also extended RV's ERCx test suite to discover these vulnerabilities automatically. Summarizing, we found that none of the listed tokens is vulnerable to the second risk. However, we stress that some tokens are vulnerable to the third issue because they possess a minting capability. See scenario for details:

Token	ERCx Report
DAI	https://ercx.runtimeverification.com/token/0x6B175474E89094C44Da98b954EedeAC495271d0F
USDC	https://ercx.runtimeverification.com/token/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48
USDT	https://ercx.runtimeverification.com/token/0xdAC17F958D2ee523a2206206994597C13D831ec7
FRAX	https://ercx.runtimeverification.com/token/0x853d955acef822db058eb8505911ed77f175b99e

LUSD	https://ercx.runtimeverification.com/token/0x5f98805A4E8be255a32880FDeC7F6728C6568bA0
BUSD	https://ercx.runtimeverification.com/token/0x4fabb145d64652a948d72533023f6e7a623c7c53
sUSD	https://ercx.runtimeverification.com/token/0x57ab1ec28d129707052df4df418d58a2d46d5f51
TUSD	https://ercx.runtimeverification.com/token/0x0000000000085d4780B73119b644AE5ecd22b376
wstETH	https://ercx.runtimeverification.com/token/0x7f39c581f595b53c5cb19bd0b3f8da6c935e2ca0
wETH	https://ercx.runtimeverification.com/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2

Scenario

1. Assume that Alice is malicious user with permissions to execute the mint function of a underlying market token.
2. Alice mints herself 2^256-1 tokens.
3. Alice executes a supply action immediately followed by a withdraw action on Morpho.
4. After the withdrawal the approved amount of AAVE on behalf of Morpho is 0.
5. Consequently, all actions that depend on AAVE pulling in tokens will revert.

Classification

The issue is classified as High severity because it can lead to a permanent deadlock of some protocol operations. The issue is classified as high difficulty because only privileged accounts who can control the total supply of the underlying reserve can execute this attack in practice. We stress that the difficulty may be lower for tokens that implement permission-less flashminting a flashborrowing functions. For example, DAI is flashmintable, however, it's not at risk because the maximum mintable amount is capped at a value at which the described attack is still not practical.

Recommendation

Consider removing the infinite approvals and moving towards exact approvals. Whenever you expect AAVE to pull in assets from Morpho, make approval of just the same amount - and not more.

1. The mitigation reduces the risk of the first scenario, as the attacker cannot arbitrarily drain idle supplies from Morpho. However, notice that the attacker might be able to move the idle supplies to AAVE by other means, at which point he might still be able to drain them.
2. Furthermore, the mitigation will allow the creation of markets for tokens that embody balance checks before approvals.
3. The third scenario is prevented because each pull of money from AAVE is independently approved.

The downside of this mitigation is that it is more gas-intensive because more approved actions are needed.

The third scenario can also be prevented by implementing a "top-up" function to increase AAVEs allowance back to a non-critical value. This mitigation does not prevent the scenario but only provides a recovery method.

Furthermore, we recommend checking vulnerabilities and edge-cases of each token for which Morpho Labs intends to create a market.

A04: Morpho can end-up in AAVE’s isolation mode

[Severity: High | Difficulty: High | Category: Security]

Morpho Labs asked Runtime Verification to look into an issue previously identified during an audit performed by Spearbit auditors.

See: spearbit-audits/review-morpho-Aave-v3#10

The auditors discovered a scenario where the Morpho contract could end up in AAVE's isolation mode. In this mode, borrowing is restricted to assets approved by AAVE's POOL_ADMIN or RISK_ADMIN role. To determine if a user is in isolated mode, AAVE checks if some specific reserve has a debt ceiling.

In the following, we refer to the reserve asset used in this check as the pivot asset.

The pivot asset

When determining if a user is in isolation mode, AAVE only checks the debt ceiling of the pivot asset. The pivot asset varies for each user according to the following rules:

1. Initially, all reserves are ordered according to their initialization time. The oldest reserve occupies the first position, and the youngest reserve the last position.
2. When a `POOL_ADMIN` drops a reserve, its position becomes free.
3. When a new reserve is initialized, it occupies the first free position.
4. Only reserves marked as collateral reserves in the user configuration are candidates for the pivot element.
5. The first reserve among all candidates in the described ordering is the pivot asset.

Collateral assets

One requirement to put Morpho into isolation mode is that the isolated reserve is marked as collateral in Morpho's user configuration on Aave.

According to Spearbit, this can happen in the following situations:

> *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.

However, we found that transfers aTokens and supply-actions of an isolated reserve don't automatically mark the reserve as collateral. AAVE checks that a reserve has no debt ceiling before marking it as collateral automatically after a transfer or supply action. Hence, the first two scenarios are not sufficient to put Morpho into isolation mode.

Below we discuss modifications to the scenarios that we think are still possible under some assumptions. There are at least two more potential situations that could put Morpho into isolation mode:

1. The AAVE `POOL_ADMIN` or `RISK_ADMIN` sets a debt ceiling of collateral reserve of Morpho
2. The AAVE `POOL_ADMIN` or `ASSET_LISTING_ADMIN` initializes a new reserve with a debt ceiling.

Scenario 1: Downgrading existing collateral assets to isolated assets

The AAVE `POOL_ADMIN` and `RISK_ADMIN` roles can set debt ceilings for any reserve at any time under the following requirements:

1. the total supply of corresponding aTokens is 0
2. the amount of aTokens accrued to AAVE's treasury is 0

If the modified reserve is ordered before the current pivot element, it could become the new pivot element putting Morpho into isolation mode.

To mitigate this scenario, one could set up a vault that permanently locks a small amount of aTokens. Alternatively, one could transfer some aTokens to an unused address (i.e., one without a known private key).

It's important the vault contract contains minimal logic. In particular, it should not be able to send out aTokens or burn them by withdrawing from AAVE. This excludes, for example, the Morpho main contract as a viable candidate.

The mitigation always falsifies the first requirement that is needed to set a debt ceiling on a reserve. This is under the assumption that aTokens are only burnt when withdrawing from the pool and that AAVE's internal accounting of the aToken supply is working as intended.

Scenario 2: Recycling reserve ids

The AAVE `POOL_ADMIN` and `ASSET_LISTING_ADMIN` roles can initialize new reserves at any time. Under the following circumstance, a new reserve can become the new pivot element putting Morpho into isolation mode.

1. The reserve was initialized with a debt ceiling
2. The reserve is ordered before the last pivot element according to the rules described above. This situation can occur when a reserve has been dropped from the pool before.
3. The reserve is marked as collateral in Morpho's user configuration on AAVE

This scenario is possible under the following sequence of actions:

1. AAVE lists a reserve "X" without a debt ceiling.
2. A user transfers "aX" to Morpho.
3. Since "X" has no debt ceiling, it passes AAVEs collateral validation. The reserve is marked as collateral in Morpho's user configuration.
4. AAVE drops "X".
5. AAVE lists a reserve "Y" with a debt ceiling.
6. The reserve takes the former position of reserve "X".
7. AAVE does not clean up user configurations when assets are dropped.
8. Since "Y" has recycled the former id of "X", it is now incorrectly marked as collateral in Morpho's user configuration.

Classification

The issue is classified as high severity because it can lead to a permanent deadlock of some protocol operations. The issue is classified as high difficulty because only privileged user’s in control of the underlying AAVE pool can create the conditions that make this attack possible.

Status

Morpho addressed the issue by implementing privilged functions that allow the governance to disable specific assets as collateral.

A05: WETHGateway: Save gas by using direct approve

[Severity: Informative | Difficulty: - | Category: Gas optimization]

The WETHGateway contract uses safeApprove from solmate's SafeTransferLib to approve WETH. The SafeTransferLib is good practice when dealing with tokens, that do not return boolean values on transfers/approves, e.g., USDT. However, it's not needed in the case of WETH because it correctly returns true on successful approve-calls. Hence, one could save some gas by calling approve directly.

Classification

The issue is classified as informative because it does not affect the contract security. Difficulty is not applicable on this issue.

Status

Morpho decided to stay with `safeApprove` function for consistency. The gas overhead is acceptable.

A06: RewardsManager: Unused variable _POOL

[Severity: Informative | Difficulty: - | Category: Best practices]

The rewards manager has an immutable `_POOL` variable, which apparently should point to the underlying AAVE pool. However, this relationship between the Morpho and AAVE instances is not enforced during deployment. Moreover, the _POOL variable is never used by the RewardsManager.

Classification

The issue is classified as informative because it does not affect the contract security. Difficulty is not applicable on this issue.

Recommendation

Consider removing the `_POOL` variable from the RewardsManager. If this is impossible or not wanted, consider adding a validation check in the constructor to ensure that `_pool == _MORPHO.POOL()`.

Status

Morpho addressed the issue by removing the unused variable.

A07: RewardsManager: Missing NatSpec comment

[Severity: Informative | Difficulty: - | Category: Best practices]

The following function of the RewardsManager lacks a NatSpec comment for the reward parameter.

```
function getUserAccruedRewards(address[] calldata assets, address user, address reward)
```

Classification

The issue is classified as informative because it does not affect the contract security. Difficulty is not applicable on this issue.

Status

Morpho Labs addressed the issue by adding said NatSpec comment.

A08: Potential deadlock when disabling reserves as collteral

[Severity: Mid | Difficulty: High | Category: Best practices]

In order to workaround an issue with AAVE reserves having loan-to-value of 0, Morpho Labs introduced to permissioned functions allowing the governance to mark reserves as (non-)collateral on AAVE and Morpho.

The function `setAsssetIsCollateral` (un)sets the underlying reserve as collateral on Morpho. This only works if the reserve is not marked as collateral in Morpho's user configuration on AAVE.

The function `setAssetIsCollateralOnPool` (un)sets an asset as collateral in Moprho's user configuration on AAVE. This only works, if the asset is marked as collateral on Morpho.

However, AAVE will disable an asset as collateral in a user configuration if the supply of the user drops to zero, for example due to withdrawals or liquidations. Consequently, it's possible for Morpho to end up in a deadlock, where an asset is marked as non-collateral in Morpho's user configuration but marked as collateral on Morpho.

Notice, that under normal circumstances the asset would be marked as collateral in Morpho's user configuration again, when the supply position becomes positive. Hence, there is no issue. However, this does not happen if the underlying reserve is an isolated asset or has a debt ceiling of zero.

```
/// @notice Sets the `underlying` asset as `isCollateral` on the pool.
/// @dev The following invariant must hold: is collateral on Morpho => is
collateral on pool.
function setAssetIsCollateralOnPool(address underlying, bool isCollateral)
    external
    onlyOwner
    isMarketCreated(underlying)
{
    if (_market[underlying].isCollateral) revert
Errors.AssetIsCollateralOnMorpho();

    _pool.setUserUseReserveAsCollateral(underlying, isCollateral);
}

 /// @notice Sets the `underlying` asset as `isCollateral` on Morpho.
/// @dev The following invariant must hold: is collateral on Morpho => is
collateral on pool.
```

```
function setAssetIsCollateral(address underlying, bool isCollateral)
    external
    onlyOwner
    isMarketCreated(underlying)
{
    if
(!_pool.getUserConfiguration(address(this)).isUsingAsCollateral(_pool.getRe
serveData(underlying).id)) {
        revert Errors.AssetNotCollateralOnPool();
    }

    _market[underlying].setAssetIsCollateral(isCollateral);
}
```

Status

Morpho Labs acknowledged the issue.

Re-entrancy analysis

This section summarizes the performed re-entrancy analysis. We first outline our general methodology. We used slither to obtain a list of potential reentrancy vulnerabilities. Not every issue reported by slither is problematic, though. It's well known that slither reports false positives. Consequently, we reviewed the list of potential reentrancies and collected some trust assumptions for each item. Eventually, we reasoned that reentrancy is impossible under the stated assumptions. We don't draw any conclusions if the premises are violated.

We remark that all the following issues were reported by slithers reentrancy-events detector. This detector identifies potential reentrancies that can lead to events that are emitted in the wrong order. The on-chain security of the smart contract is not affected by the reordering. The danger stems from off-chain components that depend on the correct ordering. For instance, an attacker could try to exploit the reordering to spoof the web frontend to trick users into signing malicious transactions. We include a sketch of a potential attack vector in the first issue. The remaining issues don't contain these instructions for brevity. We remark that the general attack vector is analogous in all cases.

To rule out a potential reentrancy problem, we must look at all the external calls before events are emitted. We must confirm that the target does not reenter the original function for every external call. Notice that an external call might invoke further external calls. Sometimes, the target contract is not known in advance. In these cases, we must make additional assumptions about the eligible targets.

All potential reentrancy issues discussed in this section follow the same structure: We first include the original report generated by slither. We then list all external calls that are transitively reachable from the original function. Eventually, we reason about each external call individually to clarify whether it can be abused to reenter the original function.

A09: PositionManagerInternal._accountRepay

[Severity: Informative | Difficulty: - | Category: -]

Slither Report

Reentrancy in **PositionsManagerInternal._accountRepay**(address,uint256,address,uint256,Types.Indexes256)
(src/PositionsManagerInternal.sol#281-341):

External calls:

- `_updateBorrowerInDS(underlying,onBehalf,vars.onPool,vars.inP2P,false)` (src/PositionsManagerInternal.sol#300)
  - **`rewardsManager.updateUserRewards`**`(user,poolToken,formerOnPool)` (src/MorphoInternal.sol#361)

Event emitted after the call(s):

- **`Events.BorrowPositionUpdated`**`(firstUser,vars.underlying,onPool,inP2P)` (src/MatchingEngine.sol#150)
  - demoted = `_demoteSuppliers(underlying,vars.toSupply,maxIterations)` (src/PositionsManagerInternal.sol#332)
- **`Events.SupplyPositionUpdated`**`(firstUser,vars.underlying,onPool,inP2P)` (src/MatchingEngine.sol#151)
  - demoted = `_demoteSuppliers(underlying,vars.toSupply,maxIterations)` (src/PositionsManagerInternal.sol#332)

Scenario

To reorder the events an attacker would need to execute the following steps.

1. Assume that Alice is an attacker.
2. Alice calls the `repay` function.
3. No event is emitted yet.
4. Alice manages to reenter into the `repay` function.
5. The inner call emits the `BorrowPositionUpdated` and `SupplyPositionUpdated` events.
6. The outer call emits the `BorrowPositionUpdated` and `SupplyPositionUpdated` events.
7. Notice, the events from the inner call are emitted before the events of the outer call, although the outer call was started before the inner call.

External call analysis

The following list summarizes the reachable external calls:

- `rewardsManager.updateUserRewards(user,poolToken,formerOnPool)` (src/MorphoInternal.sol#361)

- `IScaledBalanceToken(asset).scaledTotalSupply()`
  `(src/RewardsManager.sol#L136)`
- `_REWARDS_CONTROLLER.getRewardsByAsset(asset)`
  `(src/RewardsManager.sol#280)`
- `_REWARDS_CONTROLLER.getAssetDecimals(asset)`
  `(src/RewardsManager.sol#L285)`
- `_REWARDS_CONTROLLER.getRewardsData(asset, reward)`
  `(src/RewardsManager.sol#401)`

rewardsManager.updateUserRewards

The possibility of this attack depends on the value of the `rewardsManager`. We assume that `rewardsManager` is an instance of the `RewardsManager` contract which was deployed and is controlled by Morpho. In orther words we assume that the `rewardsManager` is a trusted component of the Morpho protocol. We need to check that the `updateUserRewards` cannot directly or indirectly re-enter into the `_accountRepay`-function. The direct case is trivial - the function does not directly call back into the `_accountRepay` function. It remains to check that the function cannot indirectly re-enter. We therefore need to analyse the reachable external calls.

IScaledBalanceToken(asset).scaledTotalSupply()

We assume that asset points to a trusted and honestly and completently operatored instance of ScaledBalanceTokenBase.sol

The `scaledTotalSupply()` function does not invoke external calls.

_REWARDS_CONTROLLER

We assume that `_REWARDS_CONTROLLER` points to a trusted and honestly and competently operatated instance of AAVE's RewardsDistributor-contract.

- The `getRewardsByAsset` function does not invoke external calls.
- The `getAssetDecimals` function does not invoke external calls.
- The `getRewardsData` function does not invoke external calls.

Conclusion

We conclude that under the listed assumptions a reentrancy attack is impossible under the given assumptions. Hence, we classify this issue as informative.

A10: PositionManagerInternal._accountWithdraw

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **PositionsManagerInternal._accountWithdraw**(address,uint256,address,uint256,Types.Indexes256)
(src/PositionsManagerInternal.sol#344-401):

External calls:

- updateSupplierInDS(underlying,supplier,vars.onPool,vars.inP2P,false)
  (src/PositionsManagerInternal.sol#365)
  - **rewardsManager.updateUserRewards**(user,poolToken,formerOnPool)
    (src/MorphoInternal.sol#361)

Event emitted after the call(s):

- Events.**BorrowPositionUpdated**(firstUser,vars.underlying,onPool,inP2P)
  (src/MatchingEngine.sol#150)
  - demoted = _demoteBorrowers(underlying,vars.toBorrow,maxIterations)
    (src/PositionsManagerInternal.sol#394)
- Events.**SupplyPositionUpdated**(firstUser,vars.underlying,onPool,inP2P)
  (src/MatchingEngine.sol#151)
  - demoted = _demoteBorrowers(underlying,vars.toBorrow,maxIterations)
    (src/PositionsManagerInternal.sol#394)

External call analysis

The following list summarizes the reachable external calls:

- rewardsManager.updateUserRewards(user,poolToken,formerOnPool)
  (src/MorphoInternal.sol#361)
- IScaledBalanceToken(asset).scaledTotalSupply()
  (src/RewardsManager.sol#L136)
- _REWARDS_CONTROLLER.getRewardsByAsset(asset)
  (src/RewardsManager.sol#280)
- _REWARDS_CONTROLLER.getAssetDecimals(asset)
  (src/RewardsManager.sol#L285)
- _REWARDS_CONTROLLER.getRewardsData(asset, reward)
  (src/RewardsManager.sol#401)

We remark that the reachable external calls are the same as for `_accountRepay` function. Under the same trust assumptions none of these targets reenters the `_accountWithdraw` function. Hence, we classify this issue as informative.

A11: PositionManagerInternal._executeBorrow

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **`PositionsManagerInternal._executeBorrow`**`(address,uint256,address,address,uint256,Types.Indexes256)` `(src/PositionsManagerInternal.sol#445-456):`

External calls:

- `vars = _accountBorrow(underlying,amount,onBehalf,maxIterations,indexes)` `(src/PositionsManagerInternal.sol#453)`
  - **`rewardsManager.updateUserRewards`**`(user,poolToken,formerOnPool)` `(src/MorphoInternal.sol#361)`

Event emitted after the call(s):

- `Events.`**`Borrowed`**`(msg.sender,onBehalf,receiver,underlying,amount,vars.onPool,vars.inP2P)` `(src/PositionsManagerInternal.sol#455)`

External call analysis

The following list summarizes the reachable external calls:

- `rewardsManager.updateUserRewards(user,poolToken,formerOnPool)` `(src/MorphoInternal.sol#361)`
- `IScaledBalanceToken(asset).scaledTotalSupply()` `(src/RewardsManager.sol#L136)`
- `_REWARDS_CONTROLLER.getRewardsByAsset(asset)` `(src/RewardsManager.sol#280)`
- `_REWARDS_CONTROLLER.getAssetDecimals(asset)` `(src/RewardsManager.sol#L285)`
- `_REWARDS_CONTROLLER.getRewardsData(asset, reward)` `(src/RewardsManager.sol#401)`

We remark that the reachable external calls are the same as for `_accountRepay` function. Under the same trust assumptions none of these targets reenters the `_executeBorrow` function. Hence, we classify this issue as informative.

A12: PositionManagerInternal._executeRepay

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **`PositionsManagerInternal._executeRepay`**`(address,uint256,address,address,uint256,Types.Indexes256)`
`(src/PositionsManagerInternal.sol#459-470):`

External calls:

- `vars = _accountRepay(underlying,amount,onBehalf,maxIterations,indexes)`
  `(src/PositionsManagerInternal.sol#467)`
  - **`rewardsManager.updateUserRewards`**`(user,poolToken,formerOnPool)`
    `(src/MorphoInternal.sol#361)`

Event emitted after the call(s):

- `Events.`**`Repaid`**`(repayer,onBehalf,underlying,amount,vars.onPool,vars.inP2P)`
  `(src/PositionsManagerInternal.sol#469)`

External call analysis

The following list summarizes the reachable external calls:

- `rewardsManager.updateUserRewards(user,poolToken,formerOnPool)`
  `(src/MorphoInternal.sol#361)`
- `IScaledBalanceToken(asset).scaledTotalSupply()`
  `(src/RewardsManager.sol#L136)`
- `_REWARDS_CONTROLLER.getRewardsByAsset(asset)`
  `(src/RewardsManager.sol#280)`
- `_REWARDS_CONTROLLER.getAssetDecimals(asset)`
  `(src/RewardsManager.sol#L285)`

- `_REWARDS_CONTROLLER.getRewardsData(asset, reward)`
  `(src/RewardsManager.sol#401)`

We remark that the reachable external calls are the same as for `_accountRepay` function. Under the same trust assumptions none of these targets reenters the `_executeRepay` function. Hence, we classify this issue as informative.

A13: PositionManagerInternal._executeSupply

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **`PositionsManagerInternal._executeSupply(`**`address,uint256,address,address,uint256,Types.Indexes256)`
`(src/PositionsManagerInternal.sol#431-442):`

External calls:

- `vars = _accountSupply(underlying,amount,onBehalf,maxIterations,indexes)`
  `(src/PositionsManagerInternal.sol#439)`
  - **`rewardsManager.updateUserRewards`**`(user,poolToken,formerOnPool)`
    `(src/MorphoInternal.sol#361)`

Event emitted after the call(s):

- `Events.`**`Supplied`**`(from,onBehalf,underlying,amount,vars.onPool,vars.inP2P)`
  `(src/PositionsManagerInternal.sol#441)`

External call analysis

The following list summarizes the reachable external calls:

- `rewardsManager.updateUserRewards(user,poolToken,formerOnPool)`
  `(src/MorphoInternal.sol#361)`
- `IScaledBalanceToken(asset).scaledTotalSupply()`
  `(src/RewardsManager.sol#L136)`
- `_REWARDS_CONTROLLER.getRewardsByAsset(asset)`
  `(src/RewardsManager.sol#280)`
- `_REWARDS_CONTROLLER.getAssetDecimals(asset)`
  `(src/RewardsManager.sol#L285)`

- `_REWARDS_CONTROLLER.getRewardsData(asset, reward)`
  `(src/RewardsManager.sol#401)`

We remark that the reachable external calls are the same as for `_accountRepay` function. Under the same trust assumptions none of these targets reenters the `_executeSupply` function. Hence, we classify this issue as informative.

A14: PositionManagerInternal._executeWithdraw

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **`PositionsManagerInternal._executeWithdraw`**`(address,uint256,address,address,uint256,Types.Indexes256)`
`(src/PositionsManagerInternal.sol#473-484):`

External calls:

- `vars = _accountWithdraw(underlying,amount,onBehalf,maxIterations,indexes)`
  `(src/PositionsManagerInternal.sol#481)`
  - **`rewardsManager.updateUserRewards`**`(user,poolToken,formerOnPool)`
    `(src/MorphoInternal.sol#361)`

Event emitted after the call(s):

- `Events.`**`Withdrawn`**`(msg.sender,onBehalf,receiver,underlying,amount,vars.onPool,vars.inP2P)`
  `(src/PositionsManagerInternal.sol#483)`

External call analysis

The following list summarizes the reachable external calls:

- `rewardsManager.updateUserRewards(user,poolToken,formerOnPool)`
  `(src/MorphoInternal.sol#361)`
- `IScaledBalanceToken(asset).scaledTotalSupply()`
  `(src/RewardsManager.sol#L136)`
- `_REWARDS_CONTROLLER.getRewardsByAsset(asset)`
  `(src/RewardsManager.sol#280)`

- `_REWARDS_CONTROLLER.getAssetDecimals(asset)`
  `(src/RewardsManager.sol#L285)`
- `_REWARDS_CONTROLLER.getRewardsData(asset, reward)`
  `(src/RewardsManager.sol#401)`

We remark that the reachable external calls are the same as for `_accountRepay` function. Under the same trust assumptions none of these targets reenters the `_accountWithdraw` function. Hence, we classify this issue as informative.

A15: Morpho.claimRewards

[Severity: Low | Difficulty: High | Category: -]

Slither report

Reentrancy in **`Morpho.claimRewards`**`(address[],address) (src/Morpho.sol#262-279):`

External calls:

- `(rewardTokens,claimedAmounts) = _rewardsManager.claimRewards(assets,onBehalf) (src/Morpho.sol#268)`
  - **`IRewardsController(_rewardsManager.REWARDS_CONTROLLER()).claimAllRewardsToSelf(assets)`**
    `(src/Morpho.sol#269)`

Event emitted after the call(s):

- `Events.`**`RewardsClaimed`**`(msg.sender,onBehalf,rewardTokens[i],claimedAmount)`
  `(src/Morpho.sol#276)`

External call analysis

The following list summarizes the reachable external calls:

- `IRewardsController(`**`_rewardsManager.REWARDS_CONTROLLER()`**`).claimAllRewardsToSelf(assets)`
  `(src/Morpho.sol#268)`
- **`IRewardsController(`**`_rewardsManager.REWARDS_CONTROLLER()`**`).claimAllRewardsToSelf(assets)`**
  `(src/Morpho.sol#268)`
- `IScaledBalanceToken(assets[i]).getScaledUserBalanceAndSupply(user);`
  Source
- `bool success = transferStrategy.performTransfer(to, reward, amount);`

- Source

We assume that `_rewardsManager` is an instance of the `RewardsManager` contract which was deployed and is controlled by Morpho. It belongs to the set of trusted contracts. The `REWARDS_CONTROLLER` function does not re-enter the contract.

We further assume that `_REWARDS_CONTROLLER` points to a trusted and honestly and competently operatated instance of AAVE's RewardsController-contract. The function does not directly reenter the contract. However, the _RewardsController infokes further external calls.

We assume that transferStrategy is an instance of either PullRewardsTransferStrategy or StakedTokenTransferStrategy. Neither contract directly reenters into Morpho.claimRewards. However both implementations invoke further external calls. The implementations are so generic that we cannot make any more reasonable assumptions about the targets of the external calls. Without any extra constraints we cannot confidentially rule out that this reentrancy scenario is possible.

We conclude that this scenario might be possible. Hence, an attacker might be able to reorder the `RewardsClaimed` events. We classify as a low severity issue because the on-chain integrity of not affected and high difficulty because we did not identify a concrete exploit scenario.

A16:Morpho.initialize

[Severity: Informative | Difficulty: - | Category: -]

Slither report

Reentrancy in **`Morpho.initialize`**`(address,Types.Iterations)` `(src/Morpho.sol#42-55):`

External calls:

- **`_POOL.setUserEMode`**`(_E_MODE_CATEGORY_ID) (src/Morpho.sol#53)`

Event emitted after the call(s):

- `Events.`**`EModeSet`**`(_E_MODE_CATEGORY_ID) (src/Morpho.sol#54)`

External call analysis

The following list summarizes the reachable external calls:

- `_POOL.setUserEMode(_E_MODE_CATEGORY_ID) (src/Morpho.sol#53)`

We remark that the initialize-method cannot be reentered because it's guarded by a mutex.

Appendix A: Glossary

This section contains the definitions of the symbols and terms used within the report. It is assumed that all definitions are indexed by an arbitrary, but fixed underlying token $\phi \in \Phi$.

Declaration	Description
$F \in \ ]0, 1]$	Collateral factor
$s: A \to \mathbb{R}_+$	Supply positions on Aave
$b: A \to \mathbb{R}_+$	Borrow positions on Aave
$s^{Pool}: A \to \mathbb{R}_+$	User supply on Pool
$s^{P2P}: A \to \mathbb{R}_+$	User supply in P2P
$b^{Pool}: A \to \mathbb{R}_+$	User borrowed amount on pool
$b^{P2P}: A \to \mathbb{R}_+$	User borrowed amount in P2P
$S^{P2P}: \mathbb{R}_+$	Total supply in P2P
$S^{Pool}: \mathbb{R}_+$	Total supply on Pool
$B^{P2P}: \mathbb{R}_+$	Total borrowed amount in P2P
$\lambda^{P2P}: \mathbb{R}_{\geq 1}$	P2P index. Increases over time.
r^S	Pool supply rate
r^B	Pool borrow rate
$\alpha \in [0, 1]$	P2P-cursor
r^α	P2P rate without protocol fees
$\delta^S: \mathbb{R}_+$	P2P supply delta
$\delta^B: \mathbb{R}_+$	P2P borrow delta

$\rho \in [0, 1]$	Reserve factor. $\rho = 0$ no fees. $\rho = 1$ means 100% of the interest above pool rate goes to the protocol.
$r^{\rho^S} = (1 - \alpha + \rho\alpha)r^S + (\alpha - \rho\alpha)r^B$	P2P supply rate with protocol fees
$r^{\rho^B} = (1 - \rho - \alpha + \rho\alpha)r^S + (\rho + \alpha - \rho\alpha)r^B$	P2P borrow rate with protocol fees

Appendix B: Issue Severity/Difficulty Classification

This sections contains a snapshot of RVs issue classification system of the time at which this report was authored. An up-to-date version of this classification system can be found at: https://github.com/runtimeverification/security/blob/master/issue-classification.md

---

Our issues ranking system is based on two axes, *severity* and *difficulty*. Severity covers "how bad would it be if someone exploited this", and is ranked *Informative*, *Low*, *Medium*, and *High*. Difficulty is "how hard is it for someone to exploit this", and is ranked *Low*, *Medium*, and *High*.

This document is *guidance* for security ratings, and is constantly changing. The lead auditor reserves the right to change severity or difficulty ratings as needed for each situation.

In some cases, it makes sense to be more clear with the client about the *recommended action*. Recommended action can be used to make the nature of the vulnerability clear to the client, and can be *fix design*, *fix code*, or *document prominently*.

Other Ranking Systems

- Immunefi ranking

Severity Ranking

Severity refers to how bad it is if this issue is exploited. This means that the *effects* of the exploit affect the severity, but *who can do the exploit* does not.

If a given attack seems to fit multiple criteria here, use the most severe classification.

High Severity

- Permanent deadlock of some or all protocol operations.
- Loss of any non-trivial amount of user or protocol funds.
- Core protocol properties do not hold.
- Arbitrary minting of tokens by untrusted users.
- DOS attacks making the system (or any vital part of the system) unusable.

Medium Severity

- Sensible or desirable properties over the protocol do not hold, but no known attack vectors due to this ("looks risky" feeling).
- Non-responsive or non-functional system is possible, but recovery of user funds can still be guaranteed.
- Temporary loss of user funds, guaranteed to be recoverable via external algorithmic mechanism like a treasury.
- Loss of small amounts of user funds (eg. bits of gas fees) that serve no protocol purpose.
- Griefing attacks which make the system less pleasant to interact with, potentially used to promote a competitor.
- System security relies on assumptions about externalities like "valid user input" or "working monitoring server".
- Deployments are not verifiable, so that phishing attacks may be possible.

Low Severity

- Slow processing of user transactions can lead to changed parameters at transaction execution time.
- Function reverts on some inputs that it could safely handle.
- Users receive less funds than expected in pure mathematical model, but bounds on this error is very small.
- Users are not protected from obviously bad choices (eg. trading into an asset with zero value).
- System accumulates dust (eg. due to rounding errors) that is unrecoverable.

Informative Severity

- Not following best coding practices. Examples include:
    - Missing input validation or state sanity checks,
    - Code duplication,
    - Bad code architecture,
    - Unmatched interfaces or bad use of external interfaces,
    - Use of outdated or known problematic toolchains (eg. bad compiler version),
    - Domain specific code smells (eg. not recycling storage slots on EVM).
- Gas optimizations.
- Non-intuitive or overly complicated behaviors (which may lead to users and/or auditors mis-understanding the code).
- Lack of documentation, or incorrect/inconsistent documentation.
- Known undesired behaviors when the security model or assumptions do not hold.

Difficulty Ranking

Difficulty refers to how hard it is to actually accomplish the exploit. The things that increase difficulty are how *expensive* the attack is, *who can perform the attack*, and *how much control you need to accomplish the attack*. Note that when analyzing the expense difficulty of an attack, you *must* take into account flash loans.

If an attack fits multiple categories here, because of factors `X` which makes it severity `S1` and `Y` which makes severity `S2`, then you need to decide:

- Are *both* `X` and `Y` necessary to make the attack happen, then use the higher difficulty.
- If *only* one of `X` and `Y` is necessary, then use the lower difficulty.

High Difficulty

- Only trusted authorized users can perform the attack (eg. core devs).
- Performing the attack costs significantly more than how much you benefit (eg. it costs 10x to do the attack vs what is actually won).
- Performing the attack requires coordinating multiple transactions across different blocks, and can be stopped if detected early enough.
- Performing the attack requires control of the network, to delay or censor given messages.
- Performing the attack requires convincing users to participate (eg. bribe the users).

Medium Difficulty

- Semi-authorized (or whitelisted) users can perform the attack (eg. "special" nodes, like validators, or staking operators).
- Performing the attack costs close to how much you benefit (eg. 0.5x - 2x).
- Performing the attack requires coordinating multiple transactions across different blocks, but cannot be stopped if detected early enough.

Low Difficulty

- Anyone who can create an account on the network can perform the attack.
- Performing the attack costs much less than how much you benefit (eg. < 0.5x).
- Performing the attack can happen within a single block or transaction (or transaction group).
- Performing the attack only requires access to a modest amount of capital and a flash-loan system.

Recommended Action

The recommended action classification can be useful to emphasize to clients the nature of a vulnerability. These are ordered from _most_ expensive/time-consuming to _least_ expensive/time-consuming for the client, so in a sense _fix design_ is more severe than _fix code_ which is more severe than _document prominently_.

Fix Design

A bug in the design means that even correct implementation will lead to undesirable behavior. This means that code development for that feature _should_ be halted, until the design issues are addressed. There is also not much point in reviewing the code for that feature. Often times, this may result in us switching to "Design Consultation", rather than continuing with "Design Review" or "Code Review".

Fix Code

A bug in the code means that the implementation does not conform to the design. The design has already been reviewed and deemed safe and operational. A fix for the code should be applied to avoid the bug (maybe adding an extra safety check, or calculating some quantity differently). If working on the code reveals problems in the design, then this should be escalated to _fix design_.

Document Prominently

The protocol design and code work as intended, but may have unintuitive or unsafe behavior when paired with some other financial product (or integration). Then in the documentation (targeted at developers trying to integrate with the product), the client should include the assumptions that this protocol makes of the other (or vice-versa). This can be important for avoiding scenarios where each protocol on its own is safe, but there is a known way to combine them that is unsafe. If the integration problem can be avoided by adding some checks in the code, you should consider (with the client) escalating this to a _fix code_. If the integrations can be made safe altogether by reworking the system a bit, you should consider (with the client) escalating this to a _fix design_.

Examples

Inside Attack

A contract has special admin accounts controlled by the internal devs. They can take a sequence of actions which steal user funds. The client may say "we will not address this,

because we trust ourselves and the community trusts us." This should be classified as _high_ severity (loss of user funds), and _high_ difficulty (compromise of the core dev team).

Arithmetic Rounding Issue

A contract has some complicated arithmetic that has unbounded rounding errors. Any user can pick inputs to the contract to control how those rounding errors accumulate, which leads to locking up (effectively burning) assets from a protocol treasury. This should be classified as _high_ severity (loss of protocol funds). The difficulty rating of this is subjective, because it may depend on current market dynamics and contract state. If at _any_ time, the user can manipulate inputs by small amount to achieve arbitrarily large output changes, then it is _low_ difficulty. If certain conditions (reasonably believed to be outside the attacker's control) need to be met, _then_ they can exploit this with limited changes to inputs, then _medium_ difficulty.

Oracle Price Attack

A contract has an oracle feed from a trusted oracle provider which uses DEX calculated prices as part of the input aggregation. A user discovers that manipulating a specific DEXs liquidity pools has an effect on the oracle price feed. The user cannot manipulate the price feed and perform the attack in the same block, because they do not have control over when oracle price updates come in. The user manipulates the price feed by interacting with the DEX liquidity pools, then waits for the price feed to adjust parameters for a lending platform that is relying on it. Once the lending platform has consumed the updated price, the user takes out a loan at more favorable rates from the lending platform. The original capital used to manipulate the liquidity pools is withdrawn. This should be classified as _high_ severity (loss of protocol funds), and _high_ difficulty (requires large funds locked into liquidity pools over the course of the attack, which spans multiple blocks).

Missing User Protections

A contract collects fees for interaction, but it over-collects fees because it doesn't know ahead how much it will need, then attempts to reimburse the user with the extra at the end. The reimbursement procedure calculates the reimbursement incorrectly, which leads to some funds being locked as dust and the user receiving too few fees afterwards. The amount of dust accumulated is small, and the error does not break any core properties of the protocol. This should be classified as _medium_ severity (loss of user funds that serves no protocol purpose), and _low_ difficulty (happens automatically for any user).

Weird Transient Dynamics due to Rounding

A protocol has a mechanism for users depositing funds and receiving a proportional amount of the protocol's token. The proportion is determined by how many funds have been deposited thus far, and contains a division. At protocol startup, the denominator in the division is zero, leading to weird rounding issues with the initial allotments of tokens. After startup, this

denominator goes up, and the problem goes away. There is no obvious downside to this, other than the initial investors do not get the same exchange rate as expected. This should be classified as _low_ severity (no known issues with accumulated rounding errors, but seems weird), and _low_ difficulty (will happen no matter what).

Divergence of Ghost Calculation

A protocol has individual interest accumulation, which is calculated and maintained on each interaction the individual has with the protocol. Separately, a _global_ interest is tracked as the total interest earned by users, but only calculated at specific events where the global interest is used. It's profitable for a user to interact often with the contract, so their interest is compounded more frequently. Meanwhile, the global interest calculation is not keeping up with the more frequent user-level compounding because it's not maintained as often. No obvious harm comes from this divergence of the global interest from the true interest, but the global interest is used in some other calculations. This is classified as _medium_ severity (sensible property about the protocol does not hold, but no known issues), and _low_ difficulty (likely to happen on its own).

Integration Attack

Suppose we are asked to audit Uniswap V3, and we know that they will be providing a price oracle based on the current status of their liquidity pools. If the Uniswap V3 pools in question have low liquidity, they are subject to manipulation by people with sufficient capital ("whale attack"). Perhaps this allows a user to manipulate the price enough to withdraw the majority of another protocol's collateral (some protocol that relies on the price feed from Uniswap V3). This is classified as _high_ severity (permanent loss of protocol funds), and _low_ difficulty (just requires a whale to manipulate Uniswap V3 liquidity pools, and the capital to manipulate the liquidity can be withdrawn). This should also be classified as a _document prominently_, so that the Uniswap team knows to make it very clear to users of their price feed that they must watch for this issue. It can be potentially escalated to a _fix design_ if the Uniswap V3 team agrees that they will not provide price feeds for low-liquidity situations, or will provide amended price feeds.