OpenZeppelin | security

MetaMorpho v1.1 Audit

Morpho

January 10, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-08-20 To 2023-08-23
Languages	Solidity

Total Issues	9 (2 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	0 (0 resolved)
Low Severity Issues	0 (0 resolved)
Notes & Additional Information	8 (1 resolved, 1 partially resolved)

Scope

We audited the changes made to the morpho-org/metamorpho-private repository by pull request #25 up to commit bf008d6.

In scope were the following contracts:

```
src
├── MetaMorpho.sol
├── MetaMorphoFactory.sol
├── interfaces
│   └── IMetaMorpho.sol
└── libraries
    ├── ConstantsLib.sol
    └── EventsLib.sol
```

System Overview

This audit focused on some incremental changes to the `MetaMorpho` contract that have been described in our previous audit report. Most importantly, the latest version of the contract has a different mechanism for handling potential losses in order to address a possible attack.

If one of the supported Morpho markets incurs bad debt, the loss will be socialized amongst all token suppliers, including `MetaMorpho` vaults. In this scenario, the total value of the assets held by the contract could decrease. Previously, this would result in a corresponding decrease in the value of the vault shares. A user could front-run this event with a transaction that redeems their vault shares at the original price, causing the remaining shareholders to absorb the full cost. They could then optionally buy their shares back at a lower price. This attack is exacerbated if the vault shares can be borrowed with a flash loan, which would occur if they are used in any existing Morpho market either as collateral or as the loan token.

To mitigate this possibility, the new codebase never decreases the variable that tracks the total assets held by the contract due to bad debt. This means that the nominal value of each vault share never decreases, except on withdrawal and redemptions. Instead, the contract tracks the cumulative losses in a separate variable that does not affect the rest of the functionality. Although this alleviates the possibility of market manipulation, it introduces a new risk: a "bank run". Since the system is now fractional reserve, shares cannot be redeemed once the assets are depleted. To restore 100% reserves, the vault owner or another interested party could donate the missing funds to an inactive receiver address like `address(1)`. Since the assets assigned to this address could never be redeemed, the vault could never be depleted. Having said that, the system could benefit from a more explicit way to cover lost assets though, since it is not immediately clear to users whether or not the bad debt has been covered.

The codebase also introduces two other minor changes. Firstly, the owner of the `MetaMorpho` vault can change the name and symbol of the associated ERC-20 share token. The token name has also been removed from the EIP-2612 domain separator, so any signature-based approvals would not be invalidated when the token name changes. Secondly, the timelock duration can be initialized to 0 so that the contract can be configured immediately. The owner is never required to update the duration, but any change will reinstate the previous time bounds, provided that the supplied value falls between the allowed ones. Users should not engage with vaults whose timelock duration is set to 0, since that would mean that the vault is still in the setup phase and conditions may be changed immediately with no prior notice.

Security Model and Trust Assumptions

Unbounded and Indefinite risk

If the missing funds are not immediately replaced, the "bank run" possibility changes the risk profile for vault shareholders to become unbounded and indefinite. Since all the losses are attributed to the last shareholders, the worst case is limited by the absolute size of the cumulative loss. This could be a significant fraction of an individual user's position, even if it is a small percentage of the total assets held by the vault. Moreover, since the loss is not offset by future earnings, the risk persists indefinitely (unless this risk is offset by an implicit donation as described above). As an example, if a vault holding 1M USD in TVL suffered a 0.5% loss due to bad debt, a bank run could be triggered and someone who deposited 5,000 USD worth of assets could face a 100% loss of funds instead of just a 0.5% loss.

The new mechanism also introduces fragility into the ecosystem. Since some vault shares may not be redeemable, they are no longer strictly fungible. Users who deposit those shares into other protocols would passively transfer this risk to the rest of the ecosystem. For example, if the shares are used as collateral in a Morpho market, liquidators may receive tokens that are ultimately not redeemable for the underlying asset.

This poses another difficulty when developing an oracle for the vault shares price. If such an oracle is to report the same price as the vault (share price only grows), then worthless shares of a depleted vault could be used as collateral to further borrow from another market. If the oracle reports the discounted effective price, the discrepancy in price could enable problematic dynamics when interacting with other protocols or liquid on-chain pools. To improve predictability, we believe that it is worth exploring designs where the loss can be alleviated directly (instead of being alleviated implicitly through a donation to `address(1)`), and where future interest payments are used to offset any existing losses.

Incentive Incompatibility

The `MetaMorpho` vault accrues interest before every deposit and withdrawal. This means that long-term shareholders are typically indifferent to how frequently the pending interest is updated in the contract. However, the new fee mechanism exacerbates a slight incentive incompatibility.

The fee recipient receives a fraction of any positive interest and is unaffected by any loss. This means they are incentivized to trigger the interest accrual immediately before and after any bad debt is realized, to prevent the loss from reducing the interest payment. This also applies to the previous codebase, but under the new mechanism, the consequence of intentionally isolating a loss-accruing event is to permanently increase the discrepancy between the expected assets and actual reserves. However, the fee recipient would typically have an overall interest in maintaining the health of the vault, which likely mitigates this concern.

Update: *In response to the introduction, the Morpho noted:*

- *We expect front-ends to not display the raw “lostAssets” value, but instead take `address(1)`’s funds into account (in the documentation, it is the recommended method to use).*
- *A version (where future interest payments are used to offset any existing losses) which would use all the newly generated interest to cover the loss is not viable for us. This is because it would increase the chances of a bank run by a lot (a vault realizing 5% bad debt would not give interest during ~1 year!). So, only a part of the interest could be used to cover the loss. In the end, it is a bit of what happens if the vault manager takes a cut on performance and uses that to cover for the lost assets (except that it is more flexible and can, for example, go faster because the manager could use past revenues).*

Notes & Additional Information

N-01 Code Simplification

Throughout the codebase, multiple possibilities for code simplification were identified:

- The previous codebase included a utility function that was used to validate a new timelock duration. This function was removed in the new codebase, even though the bound check occurs multiple times (1, 2). This is because the first instance now also allows the initial duration to be zero, so the original function cannot be called directly. Nevertheless, we believe it is both simpler in the code and easier to reason about if this edge case is handled by only calling the `_checkTimelockBounds` function if the initial duration is non-zero.
- The `_maxWithdraw` function re-implements the logic of `_convertToAssets`, which could be called instead. This would change its interface so it does not return the new total supply and total assets, which are discarded in `maxWithdraw` anyway. The `maxRedeem` function would then need to call `_convertToShares` directly. This would cause `maxRedeem` to call `_accruedFeeAndAssets` twice (once in `_convertToAssets` and once in `_convertToShares`), which is probably acceptable if the main use case is off-chain queries.

***Update:** Partially resolved in pull request #37. The `_checkTimelockBounds` utility function was brought back to enforce a consistent check for the timelock parameter, both on the constructor and on the `submitTimelock` function.*

N-02 Vault Assumes `ERC20Metadata`

The `MetaMorpho` contract retrieves the underlying asset decimals in two places: first in the `ERC4626` constructor and then in the `MetaMorpho` constructor. However, the second retrieval assumes that the `decimals()` function exists, which is not required by the ERC-20 standard.

Consider updating the docstrings to note that the asset must implement the optional ERC-20 methods.

Update: *Acknowledged, not resolved. The Morpho team stated:*

> *This issue was not introduced by the PR in scope and we want to keep the codebase changes to a minimum.*

N-03 Duplicate Imports

In `MetaMorpho.sol`, multiple instances of duplicate imports were identified:

- `PendingUint192` is imported twice (1, 2).
- `PendingAddress` is imported twice (1, 2).

Consider removing duplicate imports to improve the overall clarity and readability of the codebase.

Update: *Acknowledged, not resolved. The Morpho team stated:*

> *This issue was not introduced by the PR in scope and we want to keep the codebase changes to a minimum.*

N-04 Inconsistent `msg.sender` Usage

The `MetaMorpho` contract mostly uses the `_msgSender` function to retrieve the caller which allows it to support meta-transactions. However, it uses `msg.sender` in one instance.

Consider using the `_msgSender` function consistently.

Update: *Acknowledged, not resolved. The Morpho team stated:*

> *This issue was not introduced by the PR in scope and we want to keep the codebase changes to a minimum.*

N-05 Redundant Event Emission

The `MetaMorpho` events are always emitted on potentially state-changing actions, whether or not the state actually changes. This could still be useful to identify that a redundant action was performed. However, the `UpdateLostAssets` event is expected to change rarely and is nevertheless emitted whenever interest is accrued.

To reduce noise and ensure that event emissions are meaningful, consider only emitting the `UpdateLostAssets` event when its value changes.

Update: *Not an issue. The lost assets may be increased during deposits and withdrawals due to rounding errors in Morpho accounting.*

N-06 Unnamed Return Values

One instance of unnamed return values was identified. The return values of the `_accruedFeeAndAssets` function are not named.

Consider naming the return values of the `_accruedFeeAndAssets` function. This would make it consistent with other functions (1, 2) that return more than one output, while also improving the consistency and readability of the codebase.

Update: *Resolved in pull request #38.*

N-07 Avoid Inline `if-else` Statements

To improve code readability and to make the codebase less error-prone, consider avoiding the usage of inline `if-else` statements.

N-08 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. For conciseness and gas savings, consider replacing `if`-`revert` statements with `require` ones.

As noted in the release announcement, this recommendation is only supported by the IR pipeline (i.e., compilation via Yul, which is consistent with the current deployment configuration).

Update: *Acknowledged, not resolved. The Morpho team stated:*

> *This issue was not introduced by the PR in scope and we want to keep the codebase changes to a minimum.*

Client Reported

CR-01 Allocator can drain the MetaMorpho vault if a future IRM queries token balance

***Note:** This issue is described in a Cantina Security Review report. The Morpho team tasked us with reviewing the issue and the introduced fix for correctness, and asked that our review be included in this audit report for visibility.*

The `Allocator` role in the `MetaMorpho` vault is responsible for distributing the portfolio and managing the risks of the vault. The `Allocator` can not supply the assets to an unauthorized market (market with `supplyCap == 0`). Thus, in the scenario where the allocator's key is breached,the vault should have limited damage. The issue lies at MetaMorpho.sol#L399:

```
if (suppliedAssets == 0) continue;
```

When an unauthorized market with `suppliedAsset == 0` is provided in `reallocate`, the function just skips instead of reverting. This creates an attack vector. Since `MetaMorpho` calls `morpho.accrueInterest` in the loop, the malicious allocator could potentially get the control flow within `reallocate` function. Currently, only `AdaptiveCurveIrm` can be used, which wouldn't give users control flow and thus prevent the potential attacks. However, let's consider two hypothetical scenarios:

1. A new IRM is deployed that queries token balances to calculate interest. (This is a reasonable setting, as a lot of IRM actually depends on token's balance.)
2. Morpho-blue allows users to permissionlessly deploy their own IRMs.

Given this assumption, the malicious allocator can do the following:

1. Deploy a fake market with a malicious callback function.
2. Triggers `reallocate` with three `Allocations`. The first and the third `Allocation` are enabled markets but in the second `Allocation`, the fake market is provided.
3. `MetaMorpho` processes each allocation:
    1. The first Withdrawal is a correct one and the vault pulls tokens from the first market.
    2. The second market is malicious. The attacker get the control flow through the IRM.

    3. In the malicious callback, the exploiter deposit to MetaMorpho. Since the tokens had been pulled in previous step, the vault's price is lower. The exploiter gets vault's shares at a low price.
4. Withdraw from the metaMorpho and get the profit.

Update: *Resolved in pull request #58 at commit d92d83d and in pull request #67 at commit e8efe5e.*

Conclusion

The code under review was not too extensive although it introduced a significant change in the way MetaMorpho vaults deal with bad debt, in case it ever occurs. After thoroughly considering all potential scenarios in which this change may have an impact, no severe security issues were found, although we provided a set of recommendations to make the code even more robust.