Metamorpho v1.1
Security Review

Cantina Managed review by:

Emanuele Ricci, Lead Security Researcher
Hacker-0m, Security Researcher

February 20, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Morpho is a trustless and efficient lending primitive with permissionless market creation.

From Feb 7th to Feb 8th the Cantina team conducted a review of metamorpho-v1.1 on PR 67. The team identified no issues.

The reviewers had the following observations:

- In the current deployed metamorpho, governance will have to vet any IRMs thoroughly otherwise (given malicious callback possibility & given atleast one allocator PK is compromised) it can have systemic risk (even in worst case, risk seems to be bounded by delta in share price, need to quantify though).
- The current fix looks good, it ensures no possibility of such re-enter on disabled markets.

In particular, the changes made in PR 67 consisted in:

1. Adding a check so that only **enabled** markets can be re-allocated (withdraw or supply). This check is performed as the very first thing at each iteration before any external call.
2. Removing `supplyCap == 0` reverting during supply as it was a "*redundant*" check. The logic reverts anyway if the supplied amount goes above the market's cap.

Assumptions:

- Already enabled Morpho Blue markets are "*safe*" and cannot re-enter/call the `MetaMorpho` vault.
- `IRM` used by the enabled Morpho Blue markets is "*safe*" and cannot re-enter/call the `MetaMorpho` vault.

The general conclusions are:

- The changes are ok.
- The code always reverts if the allocation's market is **not** enabled.
- Removal of the explicit `if (supplyCap == 0) revert ErrorsLib.UnauthorizedMarket(id);` revert during the supply branch is ok. When `supplyCap == 0` it would revert anyway when `if (supplyAssets + suppliedAssets > supplyCap) revert ErrorsLib.SupplyCapExceeded(id);` is executed given that `suppliedAssets` is **always** greater than zero and `supplyAssets` can't be negative.

Nonetheless, there are some recommendations to be considered:

- When `allocations.length == 0` revert. There's no reason to allow a fully no-op reallocation.
- When `allocation.assets == supplyAssets` you should immediately **revert** without relying on the `totalWithdrawn != totalSupplied` check. This re-allocation request represents a no-op.
- If the vault has not enough withdrawn funds to perform a `MORPHO.supply`, **revert** immediately. The supply operation could happen just because some funds donated **directly** to the `vault` have not been skimmed yet. The current implementation of the `reallocate` function won't revert if, at the end, you have performed a `withdraw` that covers the supplied amount to avoid reverting when `totalWithdrawn != totalSupplied` is executed. In general, a supply re-allocation should always be "backed" by funds withdrawn and accounted in `totalWithdrawn`.
- Supply or withdrawal multiple times to/from the same market should be prohibited.

See the points below for proofs that every possible scenario works as expected.

- **Scenario 1)** `allocation.assets == supplyAssets`:

  In this case, we would have `withdrawn = supplyAssets.zeroFloorSub(allocation.assets) == ZERO`. Because `allocation.assets == supplyAssets` this reallocation operation represents a no-op: the allocator is **not changing** the allocation of the market.

  Recommendation: this operation should be reverted because it represents a no-op.

  When `withdrawn == 0` the `reallocate` logic will try to execute the "`supply`" branch. `allocation.assets != type(uint256).max` so `suppliedAssets = allocation.assets.zeroFloorSub(supplyAssets)` which is equal to `ZERO`.

  Because of that, it will execute `if (suppliedAssets == 0) continue;` and skip to the next iteration.

- **Scenario 2)** `allocation.assets < supplyAssets`:

  In this case, `withdrawn = supplyAssets.zeroFloorSub(allocation.assets)` will be **greater** than `ZERO`. The allocator wants to reduce the position in the market and perform a withdrawal.

  We enter the `withdrawn > 0` branch and perform a `MORPHO.withdraw`:

  - If `allocation.assets == 0` the allocator aims to withdraw everything and will try to **redeem** shares instead of withdrawing assets. This has a couple of benefits:
    1. You protect the operation from unexpected reverts because of front-running actions (donations, liquidations, rounding errors and so on...).
    2. The re-allocators know that the vault's position will be **fully** cleaned.
  - Otherwise, the **exact** amount of `withdrawn` underlying will be withdrawn from the market.

  The amount of assets withdrawn (value returned by `MORPHO.withdraw`) is added to the `totalWithdrawn` accumulator variable.

- **Scenario 3)** `allocation.assets > supplyAssets`:

  In this case, `withdrawn = supplyAssets.zeroFloorSub(allocation.assets)` would be equal to `ZERO`, and we enter the `else` branch that corresponds to a supply operation into the Morpho Blue market.

  The amount of assets supplied to the market is equal to `suppliedAssets = allocation.assets == type(uint256).max ? totalWithdrawn.zeroFloorSub(totalSupplied) : allocation.assets.zeroFloorSub(supplyAssets);`.

  - If `allocation.assets == type(uint256).max` it means that the re-allocator says: allocate (`supply`) to this market everything left that has been withdrawn until now (from previous iterations) and has been already supplied by previous `supply` operations already executed by `reallocate`.

    If `totalWithdrawn == 0` it means that a supply re-allocation request has been executed **before** any withdrawal reallocation. In this case, `suppliedAssets` will be equal to zero, and the loop will skip to the next re-allocation operation.

    Otherwise, `suppliedAssets` will be what's left to be supplied: `totalWithdrawn.zeroFloorSub(totalSupplied)` (the delta between the amount of assets withdrawn but not supplied yet).

    This **could** also be equal to `ZERO` if `totalWithdrawn > 0` but `totalWithdrawn == totalSupplied`. This scenario can happen if previous supply operations have already used all the withdrawn funds. Also in this scenario, `suppliedAssets == 0` and logic skip to the next iteration because of `if (suppliedAssets == 0) continue;`.

    Recommendation: this operation should revert because it represents a no-op.

  - If `allocation.assets != type(uint256).max`, the function will try to supply what has been requested by the `allocator` that is equal to `allocation.assets.zeroFloorSub(supplyAssets)`. The cases where `allocation.assets == supplyAssets` has been already covered in the Scenario 1 above.

We have covered the basic subcases where `suppliedAssets == 0`. At this point, we can assume that `suppliedAssets > 0` and we will try to supply a non-zero amount of `underlying` into the Morpho Blue market. This is an important part of the logic because it has changed compared to the one from `MetaMorpho v1.0`

```
  uint256 supplyCap = config[id].cap;

- if (supplyCap == 0) revert ErrorsLib.UnauthorizedMarket(id);

  if (supplyAssets + suppliedAssets > supplyCap) revert ErrorsLib.SupplyCapExceeded(id);
```

Removing the `supplyCap == 0` is safe from a security perspective because the same check is already **implicitly** performed by `supplyAssets + suppliedAssets > supplyCap`. When the `supplyCap == 0`, the operation will **anyway always revert**. We know at this point that `suppliedAssets` is **always** greater than zero because otherwise the iteration would have been skipped before when `if (suppliedAssets == 0) continue;` was executed. Even if `supplyAssets == 0` (nothing has been supplied yet to the market), we know that at least `suppliedAssets >= 1`. After the check has successfully passed, the logic will supply to

MorphoBlue the requested amount by executing `MORPHO.supply`. The amount of assets supplied is added to the `totalSupplied` accumulator variable.

Edge case supply a donation: The `MetaMorpho` vault implementation does not "*directly*" handle donations. When a user supplies to the vault, the vault will try to supply the amount deposited by the users across the supported market. Only that specific amount is supplied. When `vault.asset()` is donated directly to the vault (or any token in general), anyone can in a permissionless way call `vault.skim(token)` to skim the whole amount of tokens owned by the vault.

There's an edge case where if such a donated amount is not skimmed, the allocator could use it to supply to Morpho without withdrawing first from another market.

Let's assume that the `vault` has been configured with two `USDC` Morpho Blue markets with an unlimited supply cap:

- Vault has already supplied `1000 USDC` to `mbMarket0`.
- Vault has already supplied `1000 USDC` to `mbMarket1`.
- `Alice` donates `100 USDC` directly to `vault`. No one calls `vault.skim(USDC)` before the `reallocate` call.
- The allocator calls `reallocate([marketParams: mbMarket0, assets: 1100 USDC, marketParams: mbMarket1, assets: 900 USDC])`. This operation will try to supply `100 USDC` to `mbMarket0` and withdraw `100 USDC` from `mbMarket1`.

In a "normal" scenario the above execution would have **reverted** because when the first supply operation is executed, the vault would have not enough `USDC` to get "pulled" from `MORPHO` when it executes the `USDC.transferFrom(vault, ...)` during the execution of `MORPHO.supply`. It would be possible only if the withdrawal operation had been done **before** the supply one.

But because `Alice` has already donated `100 USDC`, the `MORPHO.supply` operation won't revert. This is the timeline of the execution:

1. Supply `100 USDC` to `mbMarket0`. The `USDC` used is the one donated to the `vault` by `Alice`.
2. Withdraw `100 USDC` from `mbMarket1`. These USDC won't be used and could be later on skimmed by anyone.

The `reallocate` function won't revert because at the end `totalWithdrawn` is equal to `totalSupplied` anyway.

I don't see any security issue even in this edge case given that both the markets are **enabled** by an authed `curator` and at the end of the execution the `allocator` was able to perform an operation and there's still `100 USDC` to be skimmed.

The recommendation is to **revert** in these edge case scenarios or, in general, revert **explicitly** if the `supply` operation does not have enough withdrawn funds to be executed. This check is not performed when `allocation.assets != type(uint256).max`, the logic is relying on an **implicit** revert given by the `USDC.transferFrom` execution.

Summary:

- The changes are ok.
- The code always reverts if the allocation's market is **not** enabled.
- Removal of the explicit `if (supplyCap == 0) revert ErrorsLib.UnauthorizedMarket(id);` revert during the supply branch is ok. When `supplyCap == 0` it would revert anyway when `if (supplyAssets + suppliedAssets > supplyCap) revert ErrorsLib.SupplyCapExceeded(id);` is executed given that `suppliedAssets` is **always** greater than zero and `supplyAssets` can't be negative.