MetaMorpho

Security Review

Cantina Managed review by:
Saw-mon-and-Natalie, Lead Security Researcher **Jonah1005**, Lead Security Researcher **StErMi**, Security Researcher

November 14, 2023

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

[PROJECT DESCRIPTION HERE]

From Sep 28th - Oct 16th the Cantina team conducted a review of MetaMorpho on commit hash 2f8dec...323ad1. The team identified a total of **28** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 1
- Low Risk: 7
- Gas Optimizations: 1
- Informational: 18

3 Findings

3.1 High Risk

3.1.1 The vault will stop working if one of the Morpho Blue market used by the vault stops working because of the IRM.

Severity: High Risk

Context:

- MetaMorpho.sol
- Morpho Blue issue: User's funds will be stuck forever if an enabled IRM breaks

Description: In a Morpho Blue market, the `IRM` component is called as soon as possible when one of the following operation is executed on the market itself:

- withdraw
- supply
- borrow
- repay
- liquidate

If the `IRM` breaks, each of those function will revert, preventing those operations from being executed. When this happens at the "Morpho Blue" level, the problem is critical but is isolated only to such market (or probably to each market that relies on the same `IRM`, it could depend on different factors).

In the `MetaMorpho` context, instead, the problem is amplified because one broken Morpho Blue market could break the whole Vault, preventing any users from withdrawing from the vault (even from the `idle` supply that is not exposed to any market directly).

In particular, if one market is broken, all these functions will revert

- `reallocate` if you specify a broken market in the `withdrawn` input
- `reallocate` if you specify a broken market in the `supplied` input
- `deposit` and `mint` if there's a broken market in the `withdrawQueue` (because internally, it will call `totalAssets()`)
- `withdraw/redeem`if there's a broken market in the `withdrawQueue` (because internally, it will call `totalAssets()`)
- `setFeeRecipient` if there's a broken market in the `withdrawQueue` (because internally, it will call `totalAssets()`)
- `submitFee/acceptFee` if there's a broken market in the `withdrawQueue`
- `maxWithdraw/maxRedeem` if there's a broken market in the `withdrawQueue`
- `convertToShares/convertToAssets` (implemented by `ERC4626`) if there's a broken market in the `withdrawQueue`
- `previewDeposit/previewMint` (implemented by `ERC4626`) if there's a broken market in the `withdrawQueue`
- `previewWithdraw/previewRedeem` (implemented by `ERC4626`) if there's a broken market in the `withdrawQueue`

The big problem is that the broken market can't be removed from the `withdrawQueue`. To remove such a market, the `allocator` must call `sortWithdrawQueue` but the following conditions must be met to remove it:

1) The must be no supply shares on the market (otherwise it would mean that you are leaving funds over there)
2) The market cap must be equal to `0` (otherwise people could be able to supply but not withdraw from it)

To reset the cap to zero, the curator must execute submitCap(brokenMarket, 0). This is not an issue because MORPHO is not called inside the logic and the function can't revert because of the broken market. The problem is when you want to remove the supply position that the vault itself has on the broken market. To accomplish that, you have to call reallocate and as we have already seen, that function will indeed revert because MORPHO.withdraw will revert.

Recommendation: Morpho should study and document extensively all the possible side effects of having a broken market enabled inside the vault's queue. The owner, allocator and curator must be aware about this edge case, given that each market could rely on a IRM that has not been endorsed by Morpho, or simply it could break because of the specific market condition.

Forcing the removal of a market from the withdrawQueue is an option to restore the Vault operationality, but it has huge side effects (without withdrawing first, all the supplier funds will be seen as "lost" and marked as removed). The force removal would also mean that one actor would have enough power to remove such market even if the market still have supply or a cap > 0.

Morpho:

Cantina:

3.2 Medium Risk

3.2.1 Attackers can redistribute liquidity in MetaMorpho with flash loans and pose threats to smaller markets

Severity: Medium Risk

Context: MetaMorpho.sol#L601-L608 MetaMorpho.sol#L707-L726

Description: MetaMorpho deposits users' liquidity into the underlying morpho-blue markets whenever users deposit.

MetaMorpho would instantly deposits th MetaMorpho.sol#L707-L726

```
    function _supplyMorpho(uint256 assets) internal {
        for (uint256 i; i < supplyQueue.length; ++i) {
            Id id = supplyQueue[i];
            MarketParams memory marketParams = _marketParams(id);

            uint256 toSupply = UtilsLib.min(_suppliable(marketParams, id), assets);

            if (toSupply > 0) {
                // Using try/catch to skip markets that revert.
                try MORPHO.supply(marketParams, toSupply, 0, address(this), hex"") {
                    assets -= toSupply;
                } catch {}
            }

            if (assets == 0) return;
        }

        idle += assets;
    }
```

In the _supplyMorpho function, metaMorpho iterates through all markets in the supplyQueue. If a market reaches its cap, metaMorpho deposits the liquidity into the next market until all the liquidity is distributed.

Malicious users can exploit this feature to reallocate the liquidity within the metaMorpho. Let's assume there are 2 markets (A and B) in the supply queue with caps of 100M and 20M, respectively. Currently, there's 20M of metaMorpho's liquidity in market $A and 1M in market $B. Users can deposit 99M and withdraw it instantly to reallocate all the liquidity into market $B‘.

Allocation of the liquidity would severely impact the performance of the metaMorpho. Allowing malicious users to game the portfolio would pose a huge challenge to the allocator.

Another major concern would be the threats it poses to the underlying markets. As metaMorpho grows, liquidity becomes paramount for the underlying markets. If metaMorpho withdraws all liquidity from one market, the interest rates could be pushed to unusually high levels, endangering users of the underlying markets.

Recommendation: Some thoughts on improving this:

1. Consider to set the cap of the first market as infinite. We can treat the first market as a reservoir to safeguard other smaller markets.
2. Consider not deposit funds into underlying markets at the time users deposit.

Morpho: Dealing with these potential risks is the responsibility of the allocators. Allocators can mitigate the attacks with the correct order of markets.

3.3 Low Risk

3.3.1 Return values are not used in `try`/`catch` blocks when calculating `remaining`

Severity: Low Risk

Context:

- MetaMorpho.sol#L716-L717
- MetaMorpho.sol#L742-L743

Description: In the context above the returned values/assets from calling `withdraw` and `supply` endpoints of `MORPHO` are not used:

```
try MORPHO.{supply, withdraw}(marketParams, delta, 0, address(this), address(this)) {
   remaining -= delta;
} catch {}
```

In the general term, we only know `MORPHO` is a contract that implements `IMorpho` but we might not know the exact implementation.

Recommendation: It would be best to use the returned value of assets from these endpoints when calculating `remaining` instead of the input `delta`.

Morpho:

We acknowledge the issue

3.3.2 Allowing duplicates in the `supplyQueue` could prevent the curator to enabling new markets

Severity: Low Risk

Context: MetaMorpho.sol#L673-L676, MetaMorpho.sol#L282-L297

Description: The `setSupplyQueue` allows the `allocator` to manipulate the `supplyQueue` as long as the markets in `newSupplyQueue` have been enabled and have a `cap > 0`. The current implementation of `setSupplyQueue` allows the `allocator` to add to the `supplyQueue` duplicates, something that is not possible to do to the `withdrawQueue`.

When the `curator` wants to enable a new market, it will call the `_setCap`. Such function will revert if the `supplyQueue` or `withdrawQueue` has already reached the maximum limit of `ConstantsLib.MAX_QUEUE_SIZE` elements.

Because the `allocator` can add duplicates to the `supplyQueue`, the `MetaMoprho` could reach a state where the `supplyQueue` contains more markets (some of them will be duplicates) has reached the maximum allowed length and the `curator` won't be able to add enable new markets until the `supplyQueue` has been reduced.

Test to reproduce the issue

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import {SharesMathLib} from "@morpho-blue/libraries/SharesMathLib.sol";
import {Math} from "@openzeppelin/token/ERC20/extensions/ERC4626.sol";

import "./helpers/IntegrationTest.sol";

contract SupplyQueuePreventNewMarketTest is IntegrationTest {
    using Math for uint256;
    using MathLib for uint256;
    using SharesMathLib for uint256;
    using MarketParamsLib for MarketParams;
    using MorphoLib for IMorpho;

    MarketAllocation[] internal withdrawn;
    MarketAllocation[] internal supplied;

    function setUp() public override {
        super.setUp();

        _setCap(allMarkets[0], 100 ether);
    }

    function testSupplyQueuePreventNewMarket() public {
        // fill up the queue
        Id[] memory supplyQueue = new Id[](ConstantsLib.MAX_QUEUE_SIZE);
        for (uint256 i = 0; i < supplyQueue.length; i++) {
            supplyQueue[i] = allMarkets[0].id();
        }

        vm.prank(ALLOCATOR);
        vault.setSupplyQueue(supplyQueue);

        // The curator would like to add a new market to the vault
        MarketParams memory newVaultMarket = allMarkets[1];
        vm.prank(CURATOR);
        vault.submitCap(newVaultMarket, 100 ether);

        // warp to be able to accept it
        vm.warp(block.timestamp + vault.timelock());

        // anyone should be able to accept the cap but it will revert because we have already filled the
↪   `supplyQueue` with duplicates
        // while the withdraw queue is only equal to 1
        vm.expectRevert(ErrorsLib.MaxQueueSizeExceeded.selector);
        vault.acceptCap(newVaultMarket.id());

        assertEq(vault.supplyQueueSize(), ConstantsLib.MAX_QUEUE_SIZE);
        assertEq(vault.withdrawQueueSize(), 1);
    }
}
```

Recommendation: Morpho should not allow the `allocator` to add duplicates markets to the `supplyQueue` when the `setSupplyQueue` function is executed.

There are different possible solutions to implement such behavior that would anyway end up increasing the execution gas cost of such a function. The drawbacks of the gas cost increase are anyway lower compared to the benefits of such change:

- The `_setCap` won't revert because of the duplicates in the `supplyQueue`
- The end users will pay less gas when the `deposit` and `mint` functions are executed because the user needs to iterate over unique elements of the `supplyQueue` instead of possible duplicates.

Morpho:

Allocators are assumed to be trusted by the curator. If it can't and allocators want to prevent the curator from enabling new markets, the curator can simply request the owner to disable such allocators, and there's no rush: no funds are at risk because a market is not enabled

3.3.3 `asset<>shares` conversion rounding results and possible side effects for `redeem`, `withdraw`, `maxRedeem`, `maxWithdraw`

Severity: Low Risk

Context: MetaMorpho.sol#L480-L484, MetaMorpho.sol#L475-L477, MetaMorpho.sol#L503-L514, MetaMorpho.sol#L517-L528

Description: The following issue is very similar to the one already reported for the Morpho Blue project.

Side effect when a supplier uses `maxRedeem`/`redeem` by specifying shares amount

Because of rounding/conversion, the `maxRedeem` function will return a value that is lower compared to what the user would be really able to redeem.

Let's assume this scenario:

For the sake of the example, `MetaMorpho` has only 1 market and no fees on the accrued interest.

1) Alice deposits on MM 1000 ether of `loanToken`. `vault.balanceOf(ALICE) = 1000000000000000000000000000`
2) Bob supplies X amount of `collateralToken` to be able to borrow 100 ethers of `loanToken` on the Morpho Blue market
3) 10 days pass by to accrue interest, Bob is not liquidated
4) Jack supply on Morpho Blue 10_000 ether of `loanToken` to allow Alice to withdraw the full amount

With that example, we have that

```
aliceInitialShares = 1000000000000000000000000000
maxRedeem = vault.maxRedeem(ALICE) = 999999999999999999999002754
```

if Alice had used the value returned by `maxRedeem` she would have withdrawn less `loanToken` compared to what she would have withdrawn by using `aliceInitialShares`

In general, `maxRedeem` would always return less than the real balance of share of the user.

Test to reproduce

```solidity
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import {SharesMathLib} from "@morpho-blue/libraries/SharesMathLib.sol";
import {IERC20Errors} from "@openzeppelin/interfaces/draft-IERC6093.sol";
import {Math} from "@openzeppelin/token/ERC20/extensions/ERC4626.sol";

import "./helpers/IntegrationTest.sol";

contract SUnderflowTest is IntegrationTest {
    using Math for uint256;
    using MathLib for uint256;
    using SharesMathLib for uint256;
    using MarketParamsLib for MarketParams;
    using MorphoLib for IMorpho;

    MarketAllocation[] internal withdrawn;
    MarketAllocation[] internal supplied;

    address ALICE;
    address BOB;
    address JACK;

    uint256 aliceInitialSupplyAmount;
    uint256 aliceInitialShares;

    function setUp() public override {
        super.setUp();

        _setCap(allMarkets[0], 100_000_0000 ether);
    }

    function prepareTest(bool secondSupplierSupplyOnMM) internal {
        ALICE = makeAddr("ALICE");
        vm.prank(ALICE);
        loanToken.approve(address(vault), type(uint256).max);

        BOB = makeAddr("BOB");
        vm.startPrank(BOB);
        collateralToken.approve(address(morpho), type(uint256).max);
        loanToken.approve(address(morpho), type(uint256).max);
        vm.stopPrank();

        JACK = makeAddr("JACK");
        vm.startPrank(JACK);
        collateralToken.approve(address(morpho), type(uint256).max);
        loanToken.approve(address(morpho), type(uint256).max);
        loanToken.approve(address(vault), type(uint256).max);
        vm.stopPrank();

        // ALICE SUPPLY 1000 LOAN TOKEN ON MM
        aliceInitialSupplyAmount = 1000 ether;
        loanToken.setBalance(ALICE, aliceInitialSupplyAmount);
        vm.prank(ALICE);
        aliceInitialShares = vault.deposit(aliceInitialSupplyAmount, ALICE);

        // BOB BORROWS 100 ether of loanToken from MB
        uint256 amountBorrowed = 100 ether;
        uint256 amountCollateral = amountBorrowed.wDivDown(allMarkets[0].lltv).mulDivDown(
            ORACLE_PRICE_SCALE,
            oracle.price()
        );
        vm.startPrank(BOB);
        collateralToken.setBalance(BOB, amountCollateral);
        morpho.supplyCollateral(allMarkets[0], amountCollateral, BOB, hex"");
        morpho.borrow(allMarkets[0], amountBorrowed, 0, BOB, BOB);
        vm.stopPrank();

        // 10 days pass by to accrue some interest
        vm.warp(block.timestamp + 10 days);
        vm.roll(block.number + 1);

        // JAKE SUPPLY SOME TO ALLOW ALICE TO WITHDRAW
        uint256 amountCollateralJack = 10_000 ether;
        loanToken.setBalance(JACK, amountCollateralJack);

        if (secondSupplierSupplyOnMM) {
            // implement it
            vm.prank(JACK);
            vault.deposit(amountCollateralJack, JACK);
        } else {
            vm.prank(JACK);
            morpho.supply(allMarkets[0], amountCollateralJack, 0, JACK, hex"");
        }
    }

    function testSecondSupplierOnMBRedeem() public {
        prepareTest(false);

        // ALICE tries to withdraw everything by using MM.redeem

        // we already know the amount of shares she has minted during the deposit
        // that are equal to `aliceInitialShares`

        // we can prove that because of rounding down
        // the max redeem will output an amount that is LOWER compared to what she could redeem

        uint256 maxRedeemAlice = vault.maxRedeem(ALICE);
        assertLt(maxRedeemAlice, aliceInitialShares);


        uint256 snapshotID = vm.snapshot();

        vm.prank(ALICE);
        uint256 aliceWithdrawnAssetsFromMaxRedeem = vault.redeem(maxRedeemAlice, ALICE, ALICE);

        // in this case (unlike the one with `aliceInitialShares`) alice still owns shares
        assertGt(vault.balanceOf(ALICE), 0);

        // assert that anyway the `maxRedeem` will always output something lower compared to the balanceOf
        assertLt(vault.maxRedeem(ALICE), vault.balanceOf(ALICE));
        vm.revertTo(snapshotID);

        vm.prank(ALICE);
        uint256 aliceWithdrawnAssetsFromInitialShares = vault.redeem(aliceInitialShares, ALICE, ALICE);

        // assert that ALICE does not own any more shares
        assertEq(vault.balanceOf(ALICE), 0);


        // Assert that the withdrawn amount is lower because of rounding
        assertLt(aliceWithdrawnAssetsFromMaxRedeem, aliceWithdrawnAssetsFromInitialShares);
    }
}
```

Side effect when a supplier uses `maxWithdraw`/`withdraw` by specifying assets amount

Like for the `maxRedeem/redeem`, we have something similar for the `maxWithdraw/withdraw` use case. Let's assume we are in the same scenario described above:

For the sake of the example, `MetaMorpho` has only 1 market and no fees on the accrued interest.

1) Alice deposits on MM 1000 ether of `loanToken`. `vault.balanceOf(ALICE) = 1000000000000000000000000000`
2) Bob supplies X amount of `collateralToken` to be able to borrow 100 ethers of `loanToken` on the Morpho Blue market
3) 10 days pass by to accrue interest, Bob is not liquidated
4) Jack supply on Morpho Blue 10_000 ether of `loanToken` to allow Alice to withdraw the full amount

Now in this simple case, Alice is the only supplier of the Vault, so what Alice can withdraw should be equal to `vault.totalAssets()`

But because of rounding/conversion `vault.maxWithdraw(ALICE) < vault.totalAssets()`. If Alice tries to withdraw the whole amount that she should be able to withdraw, the `withdraw` function will revert with the `ERC20InsufficientBalance` error because the conversion from `totalAssets` to shares will produce a number of shares (to be burned) greater than the current number of shares minted by Alice during the supply process.

- If Alice tries to withdraw `vault.totalAssets()` it will revert
- if Alice attempts to withdraw `vault.maxWithdraw(ALICE)` amount of assets, she will withdraw fewer assets compared to what she should be able to. In this case, the return value of `withdraw` (the number of shares burned) will be indeed lower compared to `aliceInitialShares` (the number of shares minted during the supply process). In fact, after the operation `vault.balanceOf(ALICE) > 0`

Test to reproduce

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import {SharesMathLib} from "@morpho-blue/libraries/SharesMathLib.sol";
import {IERC20Errors} from "@openzeppelin/interfaces/draft-IERC6093.sol";
import {Math} from "@openzeppelin/token/ERC20/extensions/ERC4626.sol";

import "./helpers/IntegrationTest.sol";

contract SUnderflowTest is IntegrationTest {
    using Math for uint256;
    using MathLib for uint256;
    using SharesMathLib for uint256;
    using MarketParamsLib for MarketParams;
    using MorphoLib for IMorpho;

    MarketAllocation[] internal withdrawn;
    MarketAllocation[] internal supplied;

    address ALICE;
    address BOB;
    address JACK;

    uint256 aliceInitialSupplyAmount;
```

```
uint256 aliceInitialShares;

function setUp() public override {
    // NB_MARKETS = 1;
    super.setUp();

    _setCap(allMarkets[0], 100_000_0000 ether);
}

function prepareTest(bool secondSupplierSupplyOnMM) internal {
    ALICE = makeAddr("ALICE");
    vm.prank(ALICE);
    loanToken.approve(address(vault), type(uint256).max);

    BOB = makeAddr("BOB");
    vm.startPrank(BOB);
    collateralToken.approve(address(morpho), type(uint256).max);
    loanToken.approve(address(morpho), type(uint256).max);
    vm.stopPrank();

    JACK = makeAddr("JACK");
    vm.startPrank(JACK);
    collateralToken.approve(address(morpho), type(uint256).max);
    loanToken.approve(address(morpho), type(uint256).max);
    loanToken.approve(address(vault), type(uint256).max);
    vm.stopPrank();

    // ALICE SUPPLY 1000 LOAN TOKEN ON MM
    aliceInitialSupplyAmount = 1000 ether;
    loanToken.setBalance(ALICE, aliceInitialSupplyAmount);
    vm.prank(ALICE);
    aliceInitialShares = vault.deposit(aliceInitialSupplyAmount, ALICE);

    // BOB BORROWS 100 ether of loanToken from MB
    uint256 amountBorrowed = 100 ether;
    uint256 amountCollateral = amountBorrowed.wDivDown(allMarkets[0].lltv).mulDivDown(
        ORACLE_PRICE_SCALE,
        oracle.price()
    );
    vm.startPrank(BOB);
    collateralToken.setBalance(BOB, amountCollateral);
    morpho.supplyCollateral(allMarkets[0], amountCollateral, BOB, hex"");
    morpho.borrow(allMarkets[0], amountBorrowed, 0, BOB, BOB);
    vm.stopPrank();

    // 10 days pass by to accrue some interest
    vm.warp(block.timestamp + 10 days);
    vm.roll(block.number + 1);

    // JAKE SUPPLY SOME TO ALLOW ALICE TO WITHDRAW
    uint256 amountCollateralJack = 10_000 ether;
    loanToken.setBalance(JACK, amountCollateralJack);

    if (secondSupplierSupplyOnMM) {
        // implement it
        vm.prank(JACK);
        vault.deposit(amountCollateralJack, JACK);
    } else {
        vm.prank(JACK);
        morpho.supply(allMarkets[0], amountCollateralJack, 0, JACK, hex"");
    }
}

function testSecondSupplierOnMBWithdraw() public {
    prepareTest(false);

    // ALICE tries to withdraw everything by using MM.withdraw

    uint256 totalAssets = vault.totalAssets();
    uint256 maxWithdrawAlice = vault.maxWithdraw(ALICE);

    // Alice is the only supplier so in theory she should be able to withdraw `vault.totalAssets()`
    // Alice but she can't because for the rounding in `withdraw` the conversion will try to burn
    // more shares that alice ownes
    assertGt(totalAssets, maxWithdrawAlice);
```

```
        uint256 totalAssetsToShares = totalAssets.mulDiv(
            vault.totalSupply() + 10 ** ConstantsLib.DECIMALS_OFFSET,
            totalAssets + 1,
            Math.Rounding.Ceil
        );

        // if alice tries to withdraw what she should be able to withdraw it will revert
        // because the conversion tries to burn more shares compared to the one owned by Alice
        vm.startPrank(ALICE);
        vm.expectRevert(
            abi.encodeWithSelector(
                IERC20Errors.ERC20InsufficientBalance.selector,
                ALICE,
                vault.balanceOf(ALICE),
                totalAssetsToShares
            )
        );
        vault.withdraw(totalAssets, ALICE, ALICE);
        vm.stopPrank();

        // if she tries to withdraw the "max withdrawable" the conversion won't burn all the shares
        // and she receive less than the totalAssets
        assertGt(totalAssets, maxWithdrawAlice);

        vm.prank(ALICE);
        uint256 burnedShares = vault.withdraw(maxWithdrawAlice, ALICE, ALICE);

        assertLt(burnedShares, aliceInitialShares);

        // alice has still shares to be burned
        assertGt(vault.balanceOf(ALICE), 0);

    }
}
```

Recap

- `vault.maxRedeem` returns a quantity of shares lower compared to the one owned by the user. As a result, the user who uses the value returned by `maxRedeem` will receive less token compared to what he would be able to get.
- `vault.maxWithdraw` returns a value that is lower compared to what the users could be able to withdraw in full. The amount of burned shares (returned by the `withdraw` execution) is lower compared to the number of shares that the user has minted via the `supply` operation (the number of shares minted is returned directly by `supply`)

Recommendation: Morpho should:

1) Document these problems and warn user/integrators
2) Be aware of these limitations and handle them in the UX (when generating the Vault action) when the user needs to `redeem/withdraw`
3) Warn the integrator and provide them example/best practice on how to properly `redeem/withdraw` from the vault

Cantina:

Morpho has decided to document this behavior and side effects in the PR https://github.com/morpho-org/metamorpho/pull/306. The issue has been acknowledged.

3.3.4 Consider adding slippage protection mechanisms to the `MetaMorpho` vault

Severity: Low Risk

Context: MetaMorpho.sol#L487, MetaMorpho.sol#L495, MetaMorpho.sol#L503, MetaMorpho.sol#L517

Description: The `MetaMorpho` contract implements a Vault system that inherits from the `ERC4626` standard implementation. The EIP-4626 security considerations section states that:

> "If implementors intend to support EOA account access directly, they should consider adding an additional function call for deposit/mint/withdraw/redeem with the means to accommodate slippage loss or unexpected deposit/withdrawal limits, since they have no other means to revert the transaction if the exact output amount is not achieved."

Because of that, Morpho should consider implementing slippage protection mechanisms to the `deposit`, `mint`, `withdraw` and `redeem` functions.

Note that there are already some ongoing efforts to cover `ERC4626` vaults with those mechanisms. More information can be gathered from the following links:

- ERC-5143: Slippage Protection for Tokenized Vault
- EIP-5143: original discussion on Ethereum Magicians

Similar issues have already been reported for the Morpho Bundlers projects:

- Consider adding a slippage protection mechanism to the `MorphoBundler` contract
- `ERC4626Bundler` does not offer any slippage/min output protection or could end up reverting the operation

Recommendation: Morpho should consider adding slippage protection mechanisms to the `deposit`, `mint`, `withdraw` and `redeem` functions.

Morpho:

We acknowledge this issue because EIP-5143 is stagnant for now. Also, users are expected to deposit on MetaMorpho through the bundler which does protect from slippage (see `ERC4626Bundler does not offer any slippage/min output protection or could end up reverting the operation`)

3.3.5 `MetaMorpho owner` is allowed to set itself as the `guardian`

Severity: Low Risk

Context: MetaMorpho.sol#L246-L256

Description: The current implementation of `submitGuardian` allows the `owner` of the contract to set itself as the `guardian` of the contract.

The `guardian` should be an external actor that should be able to act independently from the `owner` and revoke proposals made by the owner.

When the `guardian == address(0)` the owner can set itself as the `guardian` without any possible way to deny such operation (because there's no `guardian` to call `revokeGuardian`)

Recommendation: The `submitGuardian` function should revert if `newGuardian == owner()`

Morpho:

We acknowledge the issue

3.3.6 `MetaMorpho` `owner` can approve proposals even if they have expired

Severity: Low Risk

Context: MetaMorpho.sol#L203

Description: The `submitTimelock` logic

```
        if (newTimelock > timelock) {
            _setTimelock(newTimelock);
        }
```

allows the `owner` of the contract to increase the `timelock` value without any timelocked approval.

If we look at the `TIMELOCK_EXPIRATION` natspec it says:

```
    /// @dev The delay after a timelock ends after which the owner must submit a parameter again.
    /// It guarantees users that the owner only accepts parameters submitted recently.
    uint256 internal constant TIMELOCK_EXPIRATION = 3 days;
```

Because the `owner` can extend the `timelock` and because the `timelock` is a global variable that is not stored in the `pendingVARIABLE` (`pendingGuardian`, `pendingCap`, `pendingTimelock`, `pendingFee`) this logic allows the `owner` to extend the `timelock` and approve a pending change **even if it has already EXPIRED**.

Here is a test to showcase the issue

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

import "./helpers/IntegrationTest.sol";

uint256 constant FEE = 0.1 ether; // 10%
uint256 constant TIMELOCK = 1 weeks;

contract STimelockTest is IntegrationTest {
    using MarketParamsLib for MarketParams;

    function setUp() public override {
        super.setUp();

        vm.prank(OWNER);
        vault.setFeeRecipient(FEE_RECIPIENT);

        _setFee(FEE);
        _setTimelock(TIMELOCK);
        _setGuardian(GUARDIAN);

        _setCap(allMarkets[0], CAP);
    }

    function testQuickTL() public {
        uint256 newFee = FEE + 1;
        uint256 elapsed = TIMELOCK + ConstantsLib.TIMELOCK_EXPIRATION + 1;

        vm.prank(OWNER);
        vault.submitFee(newFee);

        vm.warp(block.timestamp + elapsed);

        vm.expectRevert(ErrorsLib.TimelockExpirationExceeded.selector);
        vault.acceptFee();

        // We are above the EXPIRATION, but what if the owner EXTEND the timelock?
        vm.prank(OWNER);
        vault.submitTimelock(TIMELOCK + 1);

        // and now we call again
        vault.acceptFee();

        uint256 endingFee = vault.fee();
        assertEq(endingFee, newFee);
    }
}
```

Recommendation: A way to prevent this scenario could be to check if there's an existing pending approval that have been expired and disallow the increase of the timelock if we are in such a state. Unfortunately, this solution is not viable because it can't be done for `pendingCap` approvals that are of `mapping` type. The solution would also increase the complexity of the logic and add side effects that would need to be handled.

Morpho should document this use case scenario and warn the user about it.

Cantina:

With the PR `https://github.com/morpho-org/metamorpho/pull/290` Morpho has decided to remove the logic about the expiration of a pending proposal. If the vault has no guardian or the guardian does not revoke the proposal, users have anyway enough time (min `ConstantsLib.MIN_TIMELOCK`) to withdraw their funds from the vault (as long as there's enough balance in `idle` or the underlying Morpho Blue markets allows it).

3.3.7 Value of underlying markets are not capped and can endanger the whole `MetaMorpho`

Severity: Low Risk

Context: MetaMorpho.sol#L531-L537

Description: MetaMorpho is an ERC4626 vault where users' share value depends on the totalAssets. In the current implementation, MetaMorpho queries all underlying markets in the withdrawQueue and sums them up as the total assets.

`TotalAssets` would reflect whatever value the `metaMorpho` holds in the market regardless of the cap limit of the market. This would break the risk assumption and make adjusting risks difficult.

If the market's value exceeds the cap, the vault's portfolio can change and become dangerous. The market's value can exceed its cap in the following situations:

1. Someone donates the market's liquidity to the vault.
2. The market's value grows naturally due to the interest rate.

These situations can naturally occur when the underlying markets become undercollateralized.

Let's consider the following scenario: MetaMorpho is a DAI vault with a TVL of 50M and has two underlying markets.The first market is a DAI-ETH market with low-risk parameters, while the second market is a high-risk, high-reward market, such as a DAI-Doge market with high-risk parameters. The allocator determines that it is only safe to allocate 5M liquidity to the DAI-Doge market due to the associated risks.

In a scenario where the DOGE price crashes, and all the debt in the DAI-Doge market becomes undercollateralized, users attempting to withdraw liquidity from the market can follow these steps:

1. Deposits a huge amount into `metaMorpho`.
2. Supply value to DAI-DOGE market and donated to `metaMorpho`.
3. Since liquidity is provided in step 2, the user can withdraw liquidity from the market.
4. Due to the price increase of `metaMorpho` in step 2, users can withdraw more liquidity than they deposited in step 1.

Recommendation: Some thoughts on improving this:

Consider *not* absorbing donations from users. I recommend implementing an admin-only function to handle value absorption, following a structure similar to Yearn's architecture where changes in share prices should only occur within transactions executed by trusted parties.

Alternatively, we can set a cap for each market within `totalAssets`.

```
function totalAssets() public view override(IERC4626, ERC4626) returns (uint256 assets) {
    for (uint256 i; i < withdrawQueue.length; ++i) {
        uint supplyCap =
        assets += max(supplyCap, _supplyBalance(_marketParams(withdrawQueue[i])));
    }

    assets += idle;
}
```

We must be extremely cautious when users withdraw from a market whose value exceeds the cap, as the share price of `metaMorpho` would change in this scenario.

Morpho:

Cantina: The Morpho team has decided to keep bad-debt realization on Morpho-Blue. The amount of bad debt in the market would reflect on the price of the market's shares. Thus, not capping the value of a market would not lead to profitable attacks.

3.4 Gas Optimization

3.4.1 Gas optimization suggestions

Severity: Gas Optimization

Context:

- MetaMorpho.sol#L268-L269
- MetaMorpho.sol#L670-L676
- MetaMorpho.sol#L158-L159

Description: We have collected a list of suggested changes that should achieve gas optimizations:

- MetaMorpho.sol#L268-L269: The `newSupplyCap == supplyCap` check can be moved before performing the external call to `MORPHO` and revert early to save gas
- MetaMorpho.sol#L670-L676: The `supplyQueue.length` and `withdrawQueue.length` can be moved before the `.push` operation and revert if `supplyQueue.length + 1 > ConstantsLib.MAX_QUEUE_SIZE || withdrawQueue.length + 1 > ConstantsLib.MAX_QUEUE_SIZE`
- MetaMorpho.sol#L158-L159: `timelock` can potentially be cached for the good path

Recommendation: Morpho:

We acknowledge this issue since the refactor is not worth the optimizations

3.5 Informational

3.5.1 Inconsistent timestamp types

Severity: Informational

Context:

- IMetaMorpho.sol#L25
- IMetaMorpho.sol#L18
- MetaMorpho.sol#L252

Description: `uint96` is used for `PendingAddress.submittedAt`, although every other place `uint64` is used for timestamps.

This would also make it inconsistent with other timestamps for the other `struct` definitions in `IMetaMorpho.sol`. Also note in `MetaMorpho`, we have the following cast to `uint64`:

```
pendingGuardian = PendingAddress(newGuardian, uint64(block.timestamp));
```

Recommendation: Document the decision of choosing `uint96` for `PendingAddress.submittedAt`. And if it is not necessary to have the current type to have all the timestamps the same consistent type choose `uint64` for this field.

Cantina:

The recommendations have been implemented in PR `https://github.com/morpho-org/metamorpho/pull/289`

3.5.2 Rename `prevIndex` to `pickedIndex`

Severity: Informational

Context:

- MetaMorpho.sol#L313-L318

Description/Recommendation:

Might make more sense to rename `prevIndex` to `pickedIndex`.

Morpho:

Cantina:

3.5.3 Explicitly attach library functions to types whenever possible

Severity: Informational

Context:

- MetaMorpho.sol#L32-L38
- MetaMorpho.sol#L712
- MetaMorpho.sol#L738
- MetaMorpho.sol#L766
- MetaMorpho.sol#L795
- MetaMorpho.sol#L799

Description/Recommendation:

In this context the whole libraries are attached to a type, even though specific functions are only needed. Also it would be best to order the `using for` directives:

```
using Math for uint256; // only using both overloaded versions of `mulDiv`
using {
    SafeCast.toUint192,
    SharesMathLib.toAssetsDown,
    UtilsLib.zeroFloorSub // UtilsLib.min is not attached as it is used directly.
} for uint256;

using {
    MorphoLib.lastUpdate,
    MorphoLib.supplyShares,
    MorphoBalancesLib.expectedMarketBalances,
    MorphoBalancesLib.expectedSupplyBalance
} for IMorpho;

using { MarketParamsLib.id } for MarketParams;
```

1. we cannot attach `Math.mulDiv` explicitly to `uint256` as there are two versions of this function, and there is no way to specify them currently:
   - `https://github.com/ethereum/solidity/issues/13107`
   - `https://github.com/ethereum/solidity/issues/3556`
2. `min` cannot be used as a member access call `x.min(y)` since this function is both defined in `Math` and `UtilsLib` and that is why in the current implementation all the call sites are of the form `UtilsLib.min(x, y)`. Otherwise the `solc`'s `TypeChecker` would throw an error:

```
else if (possibleMembers.size() > 1)
m_errorReporter.fatalTypeError(
    6675_error,
    _memberAccess.location(),
    "Member \"" + memberName + "\" not unique "
    "after argument-dependent lookup in " + exprType->humanReadableName() +
    (memberName == "value" ? " - did you forget the \"payable\" modifier?" : ".")
);
```

Also `toUint128` is both defined in `UtilsLib` and `SafeCast` but it's not used.

Morpho:

We acknowledge the issue

3.5.4 Consider tracking the `feeRecipient` when the `AccrueFee` event is emitted

Severity: Informational

Context: MetaMorpho.sol#L820

Description: The `feeRecipient` address can change during the `MetaMorpho` lifecycle and should be tracked by the `EventsLib.AccrueFee` event emission.

Recommendation: Morpho should append the `feeRecipient` information to the `EventsLib.AccrueFee` event.

Morpho:

We believe the reasoning should be the same as for:

- `Consider appending the _msgSender() to all the events triggered during authed function execution`
- `Consider appending the msg.sender (current owner) to all the events emitted inside functions that use the onlyOwner modifier`

Namely:

- it is planned to release a graph to track all events of Morpho Blue and MetaMorpho ; easing integrators' lives when dealing with the fee and where it was accrued each time
- the features lost are: filtering events by recipient/caller
- it is always possible to find back who accrued which fee and when (by indexing the recipient/privileged addresses change events - which is actually what the graph would be doing)

Though the two issues identified above only affect privileged functions so the additional gas cost is born by the privileged role, which we may be fine with.

For the above reasons, we acknowledge this issue

3.5.5 Consider appending the `_msgSender()` to all the events triggered during authed function execution

Severity: Informational

Context:

- MetaMorpho.sol#L174
- MetaMorpho.sol#L183
- MetaMorpho.sol#L192
- MetaMorpho.sol#L208
- MetaMorpho.sol#L225
- MetaMorpho.sol#L239
- MetaMorpho.sol#L254
- MetaMorpho.sol#L276
- MetaMorpho.sol#L651
- MetaMorpho.sol#L662
- MetaMorpho.sol#L684
- MetaMorpho.sol#L699

Description: The `owner`, `curator`, `allocator`, `guardian` and `feeRecipient` of the `MetaMorpho` contract can be arbitrary changed during the lifetime of the contract and could be different compared to the one specified during the deployment of the contract at `constructor` time.

Morpho should consider appending such information to all the events emitted inside those function that are auth gated.

Recommendation: Morpho should consider appending such information to all the events emitted inside those function that are auth gated.

Bonus: where it's needed, Morpho should consider appending not only the caller, but also the previous value that will be replaced by the new one. This suggestion could be applied not only to the "live" setters but also to the `pending*` state variables.

Morpho:

- if the event can only be emitted via a privileged function that can only be call by a single privileged address, then I believe logging the sender is unnecessary and redundant
- in all other cases, I believe it is useful to log the sender

Cantina:

With the PR `https://github.com/morpho-org/metamorpho/pull/308` some events now track the `msg.sender`.

Note that functions like `accept*` can be called by anyone. In that case, the internal function `_set*` won't track the privileged user (owner, guardian, curator, ...) but an arbitrary address that could be a "normal" EOA/contract.

3.5.6 Considerations about using OpenZeppelin 5.0 in `MetaMorpho`

Severity: Informational

Context: MetaMorpho.sol

Description: The current implementation of the `MetaMorpho` project is using the freshly released `v5.0` of the OpenZeppelin framework. The OpenZeppelin 5.0 has been released on October 5th 2023 and while it's true, that has been internally audited by the OpenZeppelin Auditing team itself (see OpenZeppelin Contracts Security Center), it's worth noting that it's not as mature and battle tested as the previous stable `v4.9.x` release.

Recommendation: Morpho should evaluate which are the benefits brought by adopting the OZ v5.0 version compared to a more known, stable, battle test v.4.9.x release of the same framework.

Morpho:

We acknowledge this issue

3.5.7 Consider renaming the `_staticWithdrawMorpho` function to a more meaningful name and updating the natspec comment

Severity: Informational

Context: MetaMorpho.sol#L751-L770

Description: The `_staticWithdrawMorpho` function is used by the `_maxWithdraw` function (used by `maxWithdraw` and `maxRedeem`) to simulate how much asset the user can really withdraw from the vault given the user's share balance and each Morpho Blue market available liquidity.

To better reflect the function's logic and be more clear, Morpho should consider to:

- Rename the function name to a more clear and meaningful name. A possible suggestion could be `_simulateWithdrawMorpho`
- Change the current `@dev` comment `/// @dev Fakes a withdraw of assets from the idle liquidity and Morpho if necessary.` to something like `/// @dev Simulate a withdraw of assets from the idle liquidity and Morpho if necessary.`

Recommendation: To better reflect the function's logic and be more clear, Morpho should consider to:

- Rename the function name to a more clear and meaningful name. A possible suggestion could be `_simulateWithdrawMorpho`
- Change the current `@dev` comment `/// @dev Fakes a withdraw of` assets `from the idle liquidity and Morpho if necessary.` to something like `/// @dev Simulate a withdraw of` assets `from the idle liquidity and Morpho if necessary.`

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-org/metamorpho/pull/280

3.5.8 Consider renaming the `ErrorsLib.MissingMarket` to a more meaningful name

Severity: Informational

Context: MetaMorpho.sol#L331

Description: The `ErrorsLib.MissingMarket` error is thrown by `sortWithdrawQueue` when the `allocator` tries to remove from the queue a market that still has some shares supplied in the underlying Morpho Blue market or the `cap` of such market on the vault config is still above zero (users can supply capital to it).

The error name seems to imply that the `allocator` was trying to remove from the queue a market that is **not existing**, while in reality the error is thrown when the `allocator` attempts to remove a market that cannot be removed (because it still has supply or users can supply to)

Recommendation: Morpho should rename the `ErrorsLib.MissingMarket` to a more meaningful name that correctly represents the error scenario.

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-org/metamorpho/pull/293

3.5.9 `MetaMorpho._deposit` should rename the `owner` parameter to `receiver` to follow the same nomenclature used by OZ `ERC4626` implementation

Severity: Informational

Context: MetaMorpho.sol#L599-L608

Description: `MetaMorpho` contract inherits from the OpenZeppelin `ERC4626` and should follow the same nomenclature and style used by such contract.

For that reason, the `MetaMorpho._deposit` function should rename the `owner` input parameter to `receiver`

Recommendation: Rename the `owner` input parameter of `MetaMorpho._deposit` to `receiver`

```diff
    /// @inheritdoc ERC4626
    /// @dev Used in mint or deposit to deposit the underlying asset to Morpho markets.
-    function _deposit(address caller, address owner, uint256 assets, uint256 shares) internal override {
+    function _deposit(address caller, address receiver, uint256 assets, uint256 shares) internal override {
-        super._deposit(caller, owner, assets, shares);
+        super._deposit(caller, receiver, assets, shares);

        _supplyMorpho(assets);

        // `newTotalAssets + assets` cannot be used as input because of rounding errors so we must use
↪    `totalAssets`.
        _updateLastTotalAssets(totalAssets());
    }
```

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-org/metamorpho/pull/276

3.5.10 `revokeGuardian` should document that a malicious guardian could disrupt the proposal flow without a way for the `owner` to revoke the guardian itself

Severity: Informational

Context: MetaMorpho.sol#L408-L413

Description: Once a `guardian` has been configured, the `submitGuardian` won't allow the `owner` to instantly change the `guardian` state variable.

The `guardian` role can revoke any proposal made to the `pendingGuardian`, `pendingTimelock`, `pendingCap`. This means that once a `guardian` has been configured, he can disrupt the "evolution" of the vault without any possibility for the `owner` to revoke it from the role.

In practice, the guardian can deny:

- The proposal of a new guardian (or removal if `submitGuardian(address(0))` was executed)
- The addition of new markets to the vault
- The increase of the timelock

The only option for the `owner` is to deploy a new Vault and ask the suppliers to migrate all the funds to the new one. This could be a non-trivial operation, given that some of the funds supplied by the suppliers could be non-withdrawable if they have been borrowed in the underlying Morpho Blue vault.

Recommendation: Morpho should document this scenario and let both the `owner` and `suppliers` be aware that:

- once configured, the `guardian` can deny the proposal of change of the guardian itself
- a malicious guardian could disrupt the vault "evolution"

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-org/metamorpho/pull/291`

3.5.11 The `idle` value change is not tracked by any event emission

Severity: Informational

Context: MetaMorpho.sol#L390, MetaMorpho.sol#L724, MetaMorpho.sol#L395, MetaMorpho.sol#L730

Description: The current implementation of the `MetaMorpho` contract is both explicitly and implicitly tracking the "movement" of the funds from a market to another but is not tracking with an explicit event changes made to the `idle` supply.

Recommendation: Tracking the changes made to the `idle` supply (how much has been added, how much has been removed and in general, which operation triggered the change) could be an important event to track for both dApps and monitoring tools.

Morpho:

We acknowledge this issue because the vault's overall exposure can be calculated using Morpho Blue's Supply/Withdraw events and thus idle can be deduced

3.5.12 The `reallocate` function does not emit any events

Severity: Informational

Context: MetaMorpho.sol#L343-L397

Description: The `reallocate` function changes the supply distribution between the markets and could update the `idle` variable that represents the amount of supply that remains idle inside the vault.

The current implementation of the `reallocate` function does not emit any events that would track

- From which market an `amount` of supply has been withdrawn
- From which market an `amount` of supply has been supplied
- How much the asset has been added or removed from the `idle` supply

Recommendation: Morpho should consider triggering an event for each market supply manipulation (specifying if it has been supplied or withdrawn and how much has been supplied/withdrawn) and a specific final event that would track how much has been added or removed from the `idle` supply.

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-org/metamorpho/pull/283`

3.5.13 `reallocate` can be front-runned by a donation and make it revert because of supply cap exceed

Severity: Informational

Context: MetaMorpho.sol#L382

Description: The `reallocate` function can be front-runned by an attacker that can `supply` on behalf of the `vault` and supply as much needed to make `_supplyBalance(allocation.marketParams) > supplyCap` trigger the revert.

The likelihood depends mostly on the delta between the `supplyCap` and the current asset in the market, but it's still doable and in theory, the attacker just needs to have enough funds to make it revert.

Recommendation: The DDoS attack can't be prevented, and the `allocator` have to re-structure the `reallocate` execution to withdraw first from the "capped" market and supply to it.

Morpho should document this possibility in the `reallocate` natspec documentation.

Morpho:

Cantina:

3.5.14 Revoke functions should revert if there are no pending changes to be revoked

Severity: Informational

Context: MetaMorpho.sol#L401-L406, MetaMorpho.sol#L408-L413, MetaMorpho.sol#L415-L420

Description: All the three `revoke*` functions allow the `guardian` to execute such function even if there's no pending proposal to be revoked. Allowing such behavior has two side effects:

- The `guardian` waste gas for the execution of the transaction even if such transaction won't change the state variable
- The `Revoke*` event is emitted even if nothing has been revoked. These events are usually monitored by monitoring systems and should be triggered only when the state has really changed.

Recommendation: Morpho should revert the `revoke*` functions execution if there's no pending proposal to be removed.

```
    /// @notice Revokes the `pendingTimelock`.
    function revokeTimelock() external onlyGuardian {
+       if (pendingTimelock.submittedAt == 0) revert ErrorsLib.NoPendingValue();

        emit EventsLib.RevokeTimelock(_msgSender(), pendingTimelock);

        delete pendingTimelock;
    }

    /// @notice Revokes the `pendingGuardian`.
    function revokeGuardian() external onlyGuardian {
+       if (pendingGuardian.submittedAt == 0) revert ErrorsLib.NoPendingValue();

        emit EventsLib.RevokeGuardian(_msgSender(), pendingGuardian);

        delete pendingGuardian;
    }

    /// @notice Revokes the pending cap of the market defined by `id`.
    function revokeCap(Id id) external onlyGuardian {
+       if (pendingCap[id].submittedAt == 0) revert ErrorsLib.NoPendingValue();

        emit EventsLib.RevokeCap(_msgSender(), id, pendingCap[id]);

        delete pendingCap[id];
    }
```

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-org/metamorpho/pull/225. Please note that the PR includes additional changes that were out of the scope of the issue

- The `owner` now is also considered a `guardian` and can execute all the functions that have the `onlyGuardianRole` modifier
- `revokeCap` can now be executed also (in addition to the `guardian`) by the `curator` and `owner`

3.5.15 Consider renaming the function `revoke*` to `revokePending*` and the event `*Revoked` to `Pending*Revoked`

Severity: Informational

Context: MetaMorpho.sol#L401-L406, MetaMorpho.sol#L408-L413, MetaMorpho.sol#L415-L420

Description: Inside the `MetaMorpho` contract there are three functions that allow the `guardian` to revoke (delete) the pending value of the `timelock`, `guardian` or market cap (specific to the market ID).

The current name used for those function could be seen as misleading and should be changed to reflect what the function really do. The same thing should be done for the event emitted inside the function.

Recommendation: Morpho should change the name of the `revoke*` functions and the event emitted inside of them to be more clear and reflect what the function does.

```diff
    /// @notice Revokes the `pendingTimelock`.
-   function revokeTimelock() external onlyGuardian {
+   function revokePendingTimelock() external onlyGuardian {
-       emit EventsLib.RevokeTimelock(_msgSender(), pendingTimelock);
+       emit EventsLib.RevokePendingTimelock(_msgSender(), pendingTimelock);

        delete pendingTimelock;
    }

    /// @notice Revokes the `pendingGuardian`.
-   function revokeGuardian() external onlyGuardian {
+   function revokePendingGuardian() external onlyGuardian {
-       emit EventsLib.RevokeGuardian(_msgSender(), pendingGuardian);
+       emit EventsLib.RevokePendingGuardian(_msgSender(), pendingGuardian);

        delete pendingGuardian;
    }

    /// @notice Revokes the pending cap of the market defined by `id`.
-   function revokeCap(Id id) external onlyGuardian {
+   function revokePendingCap(Id id) external onlyGuardian {
-       emit EventsLib.RevokeCap(_msgSender(), id, pendingCap[id]);
+       emit EventsLib.RevokePendingCap(_msgSender(), id, pendingCap[id]);

        delete pendingCap[id];
    }
```

Note: the whole `cap` concept could be renamed (everywhere) to `supplyCap` to specify that such limitation is only applied to the amount supplied to a market and not withdrawn from it.

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-org/metamorpho/pull/305`

3.5.16 Comments or Natspec documentation issues: missed parameters, typos or suggested updates

Severity: Informational

Context:

- MetaMorpho.sol#L283
- MetaMorpho.sol#L299-L300
- MetaMorpho.sol#L305
- MetaMorpho.sol#L301-L303
- IMetaMorpho.sol
- ErrorsLib.sol#L26
- MetaMorpho.sol#L762-L766

Description: We have found different natspec documentation issues that include missing parameters, typos or in general suggestion to better improve them.

- MetaMorpho.sol#L283: The `@dev` natspec says that the `supplyQueue` can be a "a set containing duplicate markets". The definition of "set" does not allow having duplicates in the collection. The comment must be re-written, if the queue allows having duplicates, it cannot be defined as a **set**.
- MetaMorpho.sol#L299-L300: A permutation operation is not allowed to remove or add elements to the collection. Morpho should update the comment to reflect the behavior of the `sortWithdrawQueue` logic.
- MetaMorpho.sol#L305: The `sortWithdrawQueue` allows the `allocator` not only to sort the `withdrawQueue` but also remove elements from the collection. The concept of `sort` does only allow the caller to shuffle elements following the logic of an algorithm, but it won't remove or add elements. Morpho should change the function's name to reflect the logic and outcome.

- MetaMorpho.sol#L301-L303: The `@notice` comment is not totally correct. The `allocator` can remove the market from the queue **only if** both the current supply (on Morpho Blue) is equal to `0` (shares) **AND** if the `cap == 0`. This means that even if there's no supply on the market (let's say that it has been just enabled by the `curator`), the `allocator` can't remove it because the `cap > 0` (it's the basic condition for a market to be considered "enabled")
- IMetaMorpho.sol: The natspec for the external/public function in the `MetaMorpho` should be moved to `IMetaMorpho.sol`and replaced with the `/// @inheritdoc IMetaMorpho` statement. Morpho should complete the current natspec documentation that does not fully cover all the `@param` or `@return` statements.
- ErrorsLib.sol#L26: the `underlyin` typo should be replaced with `underlying`
- MetaMorpho.sol#L762-L766: the statement "The vault withdrawing from Morpho cannot fail because" is not true. The function can indeed revert if `marketParams` is a broken market (because of the IRM). Morpho should update the comment to reflect the issue's scenario.

Recommendation: Morpho should consider fixing all the listed points to provide a better natspec documentation.

Cantina:

The recommendations have been partially implemented in the PR `https://github.com/morpho-org/metamorpho/pull/294`.

3.5.17 Consider replacing `safeIncreaseAllowance` with `forceApprove` in `MetaMorpho.constructor`

Severity: Informational

Context: MetaMorpho.sol#L121

Description: When the `SafeERC20.safeIncreaseAllowance` instruction is executed, the `MetaMorpho` contract has been just deployed and this means that the `morpho` contract has for the `MetaMorpho` contract. If it had been not the case (impossible) the execution of such instruction would make the deployment reverts for an overflow error.

Recommendation: Morpho should consider replacing `SafeERC20.safeIncreaseAllowance` with `SafeERC20.forceApprove` to be more coherent with the scope and logic that wants to be executed.

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-org/metamorpho/pull/274`

3.5.18 `MetaMorpho.constructor` should validate the `morpho` address

Severity: Informational

Context: MetaMorpho.sol#L116

Description: While it's true that usually a `MetaMorpho` Vault will be deployed via the Factory contract, nothing disallows anyone to deploy such a Vault by directly deploying the contract.

Only when the vault is deployed via the Vault, we are sure that the `morpho` input parameter won't be equal to `address(0)`

Recommendation: Morpho should add a sanity check in the `MetaMorpho.constructor` and revert if `morpho == address(0)`

Cantina:

The recommendations have been implemented in PR `https://github.com/morpho-org/metamorpho/pull/275`