PUBLIC

Code Assessment

of the Morpho Vault V2 Smart Contracts

September 5, 2025

Produced for

by

Contents

1 Executive Summary 3
2 Assessment Overview 5
3 Limitations and use of report 12
4 Terminology 13
5 Open Findings 14
6 Resolved Findings 16
7 Informational 24
8 Notes 26

1 Executive Summary

Dear Morpho Team,

Thank you for trusting us to help Morpho Labs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Morpho Vault V2 according to Scope to support you in forming an opinion on their security risks.

Morpho Labs implements the second version of Morpho curated vaults. The vaults are ERC-4626 compliant and non-custodial thanks to timelocks and to the "in-kind redemption" mechanism. They allow for simultaneous investments into several protocols and markets. The design of the role system allows for reasonable resilience against corrupt administrators.

The most critical subjects covered in our audit are asset solvency, functional correctness, and precision of arithmetic operations. Asset solvency has been improved after the issues Assets Can Be Double Counted and Unallocated Vault Adapter Can Report Assets were fixed. Given our Trust Model, security regarding all the aforementioned topics is good.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	5
• Code Corrected	4
• Risk Accepted	1
Low-Severity Findings	5
• Code Corrected	1
• Specification Changed	4

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Morpho Vault V2 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	19 May 2025	77aa7c5baa823697ed7a35fa0df2fd0f3617be00	Initial Version
2	7 Jul 2025	6431b81a9beda1e80b1091cccc1704ad9b178327	First fixes
3	6 Aug 2025	edf4606b67e27b3da2f812405edb84750da94ebf	Second fixes
4	11 Aug 2025	ce661d820fb29307981f75eb42393db1c6e42758	Third fixes
5	5 Sep 2025	275c2f9cbe6d79e8e4712497dc3cb51cc70f786f	Final fixes

For the solidity smart contracts, the compiler version `0.8.28` was chosen.

The following files are in scope of this review:

```
src:
    VaultV2.sol
    VaultV2Factory.sol
    adapters:
        MetaMorphoAdapter.sol
        MetaMorphoAdapterFactory.sol
        MorphoBlueAdapter.sol
        MorphoBlueAdapterFactory.sol
    imports:
        MorphoImport.sol
    libraries:
        ConstantsLib.sol
        ErrorsLib.sol
        EventsLib.sol
        MathLib.sol
        SafeERC20Lib.sol
        periphery:
            VaultV2AddressLib.sol
    vic:
        ManualVic.sol
        ManualVicFactory.sol
```

After V2, the scope was updated as follows:

Renamed:

```
src:
    adapters:
        MetaMorphoAdapter.sol           -> MorphoVaultV1Adapter.sol
```

```
        MetaMorphoAdapterFactory.sol    -> MorphoVaultV1AdapterFactory.sol
        MorphoBlueAdapter.sol           -> MorphoMarketV1Adapter.sol
        MorphoBlueAdapterFactory.sol    -> MorphoMarketV1AdapterFactory.sol
```

After V3, the scope was updated as follows:

Deleted:

```
src:
    libraries:
        periphery:
            VaultV2AddressLib.sol
    vic:
        ManualVic.sol
        ManualVicFactory.sol
```

2.1.1 Excluded from scope

Any file not listed above is out of the scope of this review. The protocols Morpho Vault V2 integrates with are assumed to work as expected, especially the `MetaMorpho` vaults V1.0 and V1.1 and the `MorphoBlue` markets.

2.2 System Overview

This system overview describes the refactored version (Version 3) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Morpho Labs offers Morpho Vault V2, the second version of Morpho's curated vaults. Morpho Vault V2 is a non-custodial, curated vault that allows users to deposit some underlying assets in the `VaultV2`. Optionally, a liquidity adapter can be specified. A curator can then reallocate the underlying assets to different protocols, in this case, MetaMorphoV1.0, MetaMorphoV1.1, and Morpho Blue markets. All the contracts are deployed by their dedicated factory.

2.2.1 VaultV2

The `VaultV2` is a non-custodial, curated, tokenized vault with EIP2612 support that allows users to deposit and withdraw a single asset, where critical state changes are timelocked. It offers the features of ERC4626 and implements virtual shares. The vault only supports underlying tokens implementing the `decimals()` function. The vault's decimals are at least `18`, and at most equal to the underlying asset's decimals. The timelock functionality allows the curator of the vault to submit calldata that will be subject to a timelock specific to each target function; anybody can execute the submitted call once the timelock has elapsed. Before every liquidity movement, the vault accrues interest and mints performance and management fees. The interest is limited to a `MAX_RATE`, which can be up to `200%` APR. The performance fee is taken as a percentage of the accrued interest, and the management fee is taken as a percentage of the total assets in the vault. The interest accrual and accounting of the fees is always done prior to any changes in total assets or total supply. The formulas for the fees are:

$$elapsed = block.timestamp - lastUpdate$$

$$realAssets = \sum_i adapters_i.realAssets()$$

$$maxTotalAssets = totalAssets * (1 + elapsed * maxRate)$$

$$newTotalAssets = min(realAssets, maxTotalAssets)$$

$$interest = max(newTotalAssets - totalAssets, 0)$$

$$performanceFeeAssets = \frac{performanceFee * interest}{1e18}$$

$$managementFeeAssets = \frac{managementFee * newTotalAssets}{1e18}$$

$$newTotalAssetsWithoutFees = newTotalAssets - performanceFeeAssets - managementFeeAssets$$

$$performanceFeeShares = \frac{performanceFeeAssets * (totalSupply + virtualShares)}{newTotalAssetsWithoutFees + 1}$$

$$managementFeeShares = \frac{managementFeeAssets * (totalSupply + virtualShares)}{newTotalAssetsWithoutFees + 1}$$

The liquidity is allocated through adapters (see Adapters). Each adapter returns a set of allocation IDs describing the exposure of the vault to some parameters of the adapter. It is possible to set absolute and (soft) relative caps for each ID.

If needed, the vault can be gated with three gates: an entry (`sendAssetsGate`) and an exit (`receiveAssetsGate`) gate for the underlying asset, and a gate to control the transfers of the shares (`sharesGate`). The implementations of such gates are not included in the scope of this audit, but the vault is designed to be able to support them. The `sendAssetsGate` is queried to know whether an account can send the underlying asset to the vault (`deposit(), mint()`). The `receiveAssetsGate` is queried to know whether an account can receive the underlying asset upon withdrawal/redemption (`withdraw(), redeem(), forceDeallocate()` for the penalty fee). Note that the `receiveAssetsGate` check is skipped when the recipient is the vault itself to ensure `forceDeallocate()` cannot be blocked by a gate. The `sharesGate` is queried to know whether an account can receive or transfer shares (used in `transfer(), transferFrom(), deposit(), mint(), withdraw(), redeem(), forceDeallocate(), accrueInterestView()`). If gates are used, the vault might not be non-custodial anymore.

The vault defines multiple roles: owner, curator, sentinels, and allocators.

The **owner** can set the owner and the curator and update the set of sentinels. There is only one owner.

The **curator** of the vault has access to direct and timelocked actions. The direct actions include increasing the timelock duration for a given timelocked function up to 3 weeks, decreasing the absolute and relative caps for a given allocation ID, submitting a call that will be subject to a timelock, and canceling a timelocked call while it is pending. Note that there is no minimal timelock duration, and all timelocks are set to 0 by default. The timelocked actions are:

- set/unset an address as an allocator
- update the `maxRate`. The maximum value is the per-second value of `200%` APR .
- update the `sharesGate`, `sendAssetsGate` and `receiveAssetsGate`
- set/unset an address as an adapter
- abdicate submit for a certain function. This will disable the option to submit a call for a given timelocked function, effectively disabling it
- decrease the timelock duration for a given timelocked function, except for the `decreaseTimelock` function itself, whose timelock is hardcoded to 3 weeks
- update the performance and management fees and the fee recipients. The maximum performance fee is `50%` and the maximum management fee is 5% APR.

- increase the absolute and relative caps for a given allocation ID. Note that the actual allocations can exceed the caps, as the caps are only enforced when adding to allocations.
- update the force deallocation penalty for a given adapter

The **allocators**' role is to manage the liquidity of the vault by allocating and deallocating the underlying assets to adapters. They can also set the liquidity adapter for the vault and its associated data with `setLiquidityAdapterAndData()`. The liquidity adapter also needs to be whitelisted before it can be used. If set but not whitelisted, or if the associated data is broken, joining the vault will be impossible. When allocating and deallocating liquidity, the adapters return a set of allocation IDs along with the change in allocation (positive or negative) that must be recorded (see Adapters). The reported change will be added to the allocation under the returned IDs in `VaultV2` to track the value changes of the vault. The change is usually computed as the difference between the current value of the adapter's position and its vault allocation. Any positive difference is an interest. Any negative difference is a loss. The realization of gains and losses is done whenever interest is accrued.

The **sentinels**' role is to take action in case of emergency. They can decrease the relative and absolute caps for arbitrary allocations IDs, cancel a timelocked call while it is pending, and deallocate liquidity from the adapters.

Users can interact with the vault using standard ERC4626 functions, such as `deposit()`, `mint()`, `withdraw()`, `redeem()`, `transfer()`, `transferFrom()` and `approve()`. The vault also implements the EIP2612 `permit()` function, allowing users to approve a transfer of shares using a signed message. Upon a `deposit()` or a `mint()`, the underlying asset is allocated to the liquidity adapter if it is set and stays in the vault as idle liquidity otherwise. To ensure that users can always withdraw assets, even when the liquidity market is empty or misconfigured, there is a `forceDeallocate()` function. This special function takes arrays of adapters, data for the adapters, and amounts to deallocate. To avoid any abuse of this function, a penalty fee (capped at 2%) is taken on the amount of assets deallocated for each adapter. The penalty fee is taken in shares from the `onBehalfOf` address and the shares are withdrawn to the vault. The redeemed penalty assets are accounted in the total assets. As a result, they are evenly given to all users who hold shares in the vault.

2.2.2 Adapters

Two adapters are currently implemented:

`MorphoVaultV1Adapter` can be used to connect to MorphoV1.0 and MorphoV1.1 vaults. The `VaultV2` parent vault can deposit and withdraw liquidity by calling `allocate()` and `deallocate()` on the adapter. These functions return the allocation IDs related to the adapter. It has only one ID, `adapter`, which is the adapter itself. Additionally, the change in allocation is computed as the difference between the value in underlying assets of the adapter's balance in the integrated Morpho vault and the current allocation in the `VaultV2`.

`MorphoMarketV1Adapter` can be used to connect to Morpho Blue markets. The `VaultV2` parent vault can deposit and withdraw liquidity by calling `allocate()` and `deallocate()` on the adapter. These functions return the allocation IDs related to the adapter. The IDs are `adapter`, `collateralToken`, and the market parameters of the Morpho Blue market. Additionally, the change in allocation is computed as the difference between the value in underlying assets of the adapter's position in the target Morpho Blue market and the current allocation in the `VaultV2`.

The owner of the parent vault can set a skim recipient, which can skim arbitrary ERC20 tokens from the adapters, except the `MetaMorpho` shares in the case of the `MorphoVaultV1Adapter`.

2.2.3 Changes from Version 1

This section describes the changes made to **Version 2** from **Version 1** of the codebase.

- The performance fee calculation is now exact in comparison to V1 where the performance fee shares represented slightly fewer assets than the target performance fee assets.

- If the VIC reverts in V2, the vault is DOSed until the VIC is replaced, in V1 the VIC was allowed to fail in `accrueInterestView()`.
- The maximum timelock was increased from 2 weeks to 3 weeks.
- The liquidity adapter does not need to be already whitelisted. In V1 this was required.
- The vault became fully ERC4626 compliant in V2
- The adapters now return their accrued interest when an allocation is modified. The loss is now managed via `realizeLoss()`. In V1, the adapters returned the loss when an allocation was modified.
- Sentinels are now only allowed to set the interest and max interest to 0 in the `ManualVic`, as opposed to V1 where they were able to set it up to the maximum, like an allocator.

2.2.4 Changes from Version 2

This section describes the changes made to Version 3 from Version 2 of the codebase.

- Losses are realized without delay. In V2, the losses were not directly reflected in the total assets and needed a dedicated function to realize them.
- The interest is now computed directly from the values returned by querying all adapters in a loop during `accrueInterestView()`. Version 2 was using a VIC indicating the interest to distribute per second. In Version 3 the VIC was completely removed from the codebase.
- The adapters return the change in allocation instead of either the interest on allocation movement or a loss on loss realization.
- Donations to vaults or adapters are now counted in `totalAssets`. However, the maximum interest rate ensures that the share price cannot increase too quickly due to donations (at most `200%` APR).

2.2.5 Changes in Version 5

This section describes the changes made in Version 5 of the codebase.

- The `SharesGate` has been split into two separate contracts, `ReceiveSharesGate` and `SendSharesGate`.
- The `setMaxRate` function is now callable by the allocator role instead of the curator. The timelock has been removed.
- The `abdicateSubmit()` function has been removed. Instead, the timelock of a function should now be set to uint256.max in order to disable it permanently.
- The maximum timelock duration has been removed.
- The timelock duration of `decreaseTimelock()` is now the timelock duration of the function whose timelock is being decreased (e.g. the timelock of `decreaseTimelock(setIsAdapter)` is `timelock[setIsAdapter]`). This means that any function whose timelock has been set to uint256.max can never have `decreaseTimelock()` called on it again, as the unlock time will overflow, causing a revert.
- An adapter registry can now be added to a VaultV2 to restrict the adapters. This is useful to commit to using only a certain type of adapters. If `adapterRegistry` is set to `address(0)`, the vault can have any adapters. When an adapterRegistry is set, it retroactively checks already added adapters. The invariant that adapters of the vault are all in the registry holds only if the registry cannot remove adapters (is "add only"). The `adapterRegistry` can be set by the curator with a timelock.

- The number of pending timelocked actions is tracked per function selector. It is increased on `submit()` and decreased when the function call is executed via `timelocked()` or when `revoke()` is called.

2.3 Trust Model

The vaults were designed in such a way that no privileged actor can steal user funds or prevent users from withdrawing them (unless they are blacklisted by the `receiveAssetsGate` or `sharesGate`). Mismanagement of the vault can, however, indirectly incur losses for users; the timelock mechanisms are there to allow users to react in time and limit losses.
The following roles can be identified in the system:

1. Owner. Semi-trusted. Can set the curator and the sentinels. If malicious, can acquire curator and then allocator privileges, and reallocate all the funds to a malicious adapter that steals user funds; this action will be timelocked, so users should have the time to exit.
2. Curator. Semi-trusted. Same as above, can steal funds if users do not react within the timelock delay. The curator is expected to check that the `MetaMorphoVaults` and `MorphoBlue` markets they whitelist (absolute cap `> 0`) have a safe initial liquidity (see How to Check Market Initial Liquidity for Safety) and a legitimate oracle and IRM for Morpho Blue markets.
3. Allocator. Semi-trusted. Can allocate and deallocate at will among existing whitelisted adapters and IDs. They are also expected to set the `max_rate` correctly and not deny users their yield by setting it to a too low value.
4. Sentinel. Semi-trusted. Can decrease caps and deallocate.
5. ReceiveSharesGate and SendSharesGate. If set, fully trusted. Can prevent users from receiving and transferring their shares, and SendSharesGate can prevent users from exiting the vault. If malicious, ReceiveSharesGate can DoS the entire vault by reverting on either of the fee recipients.
6. Receive assets gate. If set, fully trusted. Can prevent users from exiting the vault. If malicious, can DoS the vault exits and loss realizations by always returning false or reverting.
7. Send assets gate. If set, semi-trusted. Can prevent users from entering the vault. If malicious, can DoS the vault entries, but cannot block from exiting the system.
8. Users are untrusted.

The following assumptions are made about the configuration:

- The underlying asset of a `VaultV2` is expected to be a standard ERC20 compliant token. Tokens with special behaviors, such as fees on transfer, rebasing balances, or transfer hooks (reentrancies) are not supported.
- The underlying asset of a `VaultV2`, if it implements a blacklist functionality, is expected to never blacklist the `VaultV2`, its adapters or its adapters underlying systems.
- The underlying asset of a `VaultV2` is expected not to have very low decimals, such as `GUSD` with 2 decimals. Such low decimals might trigger incorrect accounting and value losses due to precision errors.
- A `VaultV2` must never have an adapter integrating with its parent vault, creating a dependency loop. If this happens, a loss could lead to a total loss for the vault.
- The `feeRecipient` of MorphoBlue must never be set to a `VaultV2`, as `expectedSupplyAssets()` would report an incorrect value, as pointed out by the NatSpec of the function.
- The adapters are expected to return IDs that are consistent with their integrated system.

- The integrated systems are expected to never re-enter the `VaultV2`.
- The function `borrowRateView()` of the IRM of a MorphoBlue market integrated directly or through a Vault V1 should not revert. If it does, the `VaultV2` will be DOSed as `accrueInterestView()` would revert as well.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation
- Trust: Violations to the least privilege principle

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Users Can Escape Losses Risk Accepted	
Low-Severity Findings	0

5.1 Users Can Escape Losses

Design Medium Version 1 Risk Accepted

CS-MORPHO-VLT2-001

In case they see an unrealized loss in an adapter, users can escape it by redeeming their shares before the loss is realized and buying back the shares at a lower price later. This increases the percentage loss of the other vault share holders, as they now cover the escaped loss.

Consider the following scenario where there is an unrealized loss of 10% in an adapter. User A holds 90% of all shares in the vault:

1. User A redeems all their shares.
2. The loss is realized.
3. User A deposits again (in a later transaction)

Now, user A has a 0% loss, while other users in the vault have suffered a 100% loss. Note that it is not possible to flash-borrow shares to do this, as they cannot be re-deposited in the same transaction.

This increased loss could lead to a situation similar to a bank run, as many users might try to redeem their shares before the loss is realized, leaving the remaining users to take the loss, and making the vault insolvent in the worst cases.

Risk accepted:

The Morpho Labs team is aware of this behavior and accepts the risk. They state that:

```
It is fundamental that experienced/more reactive lenders will be able to escape the loss: even if you add a delay,
those lenders could react before this delay. It is correct that it also adds an incentive for lenders to realize the loss.
Note that the same behavior exists on Morpho Blue markets, the scenario is very similar.
```

In Version 2, the loss realization logic has been changed. There is now an explicit incentive to call the newly added `realizeLoss()` function. This makes it unlikely that multiple users will be able to escape

losses at the same time. However, the user calling `realizeLoss()` can easily withdraw before realizing the loss. The escaped losses are an additional incentive to be the one calling `realizeLoss()`. As a result, the incentive to call `realizeLoss()` is not the same for all users. Those with the most shares have the greatest incentive.

In Version 3, the `realizeLoss()` function has been removed, and the loss is now realized automatically whenever interest is accrued, which happens before every deposit or withdrawal. Users can now only escape losses if they can predict that the loss will happen before it has been realized in the underlying adapter's market. This may be possible if the loss is caused by a liquidation that causes bad debt, which can be predicted.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- totalAssets Can Be Manipulated Code Corrected

High-Severity Findings	0

Medium-Severity Findings	4

- Assets Can Be Double Counted Code Corrected
- Unallocated Vault Adapter Can Report Assets Code Corrected
- Fee Inconsistency When Realizing a Loss Code Corrected
- Low-level STATICCALL Results in Reading Dirty Memory Code Corrected

Low-Severity Findings	5

- Interest Not Accrued When Set to Zero Specification Changed
- Interest Cannot Always Be Distributed Code Corrected
- Raw Decoding of STATICCALL's Return Data Does Not Check Length Specification Changed
- Sentinels Can Increase Interest Rate in ManualVic Specification Changed
- totalAssets() Is Out-Of-Sync Specification Changed

Informational Findings	5

- Event Emitted Even When State Is Not Updated Specification Changed
- Small Losses Can Be Offset by Interest Specification Changed
- Buggy VIC Can Burn All Gas in STATICCALL Context Code Corrected
- No NatSpec for Functions Code Corrected
- VIC Must Not Revert Specification Changed

6.1 totalAssets Can Be Manipulated

Design Critical Version 1 Code Corrected

CS-MORPHO-VLT2-016

If a `MorphoBlueAdapter` is registered, the function `forceDeallocate()` can be abused to force a loss on the system by leveraging the permissionlessness of Morpho Blue. The attack works as follows. The attacker:

1. Creates a Morpho Blue market with arbitrary loan and collateral tokens, and an oracle they can manipulate.
2. Creates a large supply position in the market and gifts it to the `MorphoBlueAdapter`.
3. Calls `forceDeallocate()` with `0` assets, but with the data of their market. This will register the large gifted position in `assetsInMarket[marketId]`.

4. Takes a loan in the target market.
5. Manipulates the oracle to make the borrow position heavily unhealthy and liquidates the position with a lot of bad debt.
6. Calls `forceDeallocate()` again with `0` assets and with the data of their manipulated market. This will now make the system believe it had a massive loss.

This allows the attacker to reduce the `totalAssets` at will.

This issue was reported independently by Morpho Labs during the review.

Code corrected:

The loss accounting logic was refactored in Version 2. There is now an explicit `realizeLoss()` function used to realize losses. In `MorphoMarketV1Adapter` (renamed from `MorphoBlueAdapter`), the loss is bounded by the `allocation` of the market. As a result, a market with zero `allocation` always has zero loss. Additionally, it is ensured in `VaultV2` that a market ID with an `absoluteCap` of zero can never have a non-zero `allocation`. This ensures that there can only be losses from markets that have explicitly been assigned an `absoluteCap` from the curator.

In Version 3, the `realizeLoss()` function has been removed, and the loss is now realized automatically.

6.2 Assets Can Be Double Counted

Correctness **Medium** **Version 3** **Code Corrected**

CS-MORPHO-VLT2-019

In `MorphoMarketV1Adapter`, `marketParams` are added to the `marketParamsList` in `allocate()` if the `allocation` is 0 and are removed in `deallocate()` if the `allocation` becomes 0.

However, it can happen that the `allocation` becomes 0 in `allocate()`, if it is called with zero assets, and the underlying market has suffered a total loss. In this case, the `marketParams` will not be removed from the list. The next time `allocate()` is called with non-zero assets, the same `marketParams` will be added again, leading to double counting of assets deposited to that market, which will lead to an incorrect share price.

If the `MorphoMarketV1Adapter` is the liquidity adapter, this double counting can be triggered permissionlessly by any user, given that there is a market with a non-zero allocation that has suffered a total loss.

Code corrected:

In Version 4, `marketParams` are removed from the `marketParamsList` on both `allocate()` and `deallocate()`, if the allocation becomes zero. This ensures that there cannot be any duplicates in the list.

6.3 Unallocated Vault Adapter Can Report Assets

Design | Medium | Version 3 | Code Corrected

CS-MORPHO-VLT2-020

The following requirement from the natspec can be broken:

- The `totalAssets()` calculation ignores markets for which the vault has no allocation.

If a `MorphoVaultV1Adapter` is whitelisted but has no allocation, it is still possible to donate a position of the underlying vault to the adapter. The position will count towards the `totalAssets()` (the increase over time is bounded by the max rate), but the allocation is still `0` and the adapter cannot be (force)deallocated. This goes against the requirements.

Code corrected:

In Version 4, the `MorphoVaultV1Adapter` always reports zero assets if its `allocation` is zero.

6.4 Fee Inconsistency When Realizing a Loss

Design | Medium | Version 1 | Code Corrected

CS-MORPHO-VLT2-002

The callpaths involving `allocate()` will take the management fee through `accrueInterest()` before realizing the loss (`allocate()`, `mint()`, `deposit()`). Some of the callpaths involving `deallocate()` also behave similarly (`redeem()`, `withdraw()`), but the other callpaths involving `deallocate()` will first realize the loss, and then eventually take the management fee. This means that for the same initial state, the management fee that is taken will depend on which action will be executed first.

Example:

`totalAssets = A loss = L interest = I`

Case 1 with `allocate()`:

1. `allocate()` is called
2. management fee is taken on `A + I`
3. loss is realized

Case 2 with `deallocate()`:

1. `deallocate()` is called
2. loss is realized
3. management fee is eventually taken on `A - L + I`

Code corrected:

In Version 2 the loss realization code was refactored. There is now a dedicated `realizeLoss()` function that is called to realize losses. As part of this function, `accrueInterest()` is called before the loss is realized. This ensures that the management fee is always taken on the total assets including any accrued interest before the loss is realized, thus eliminating the inconsistency.

In Version 3, the `realizeLoss()` function has been removed, and the loss is now realized automatically whenever interest is accrued. As a result, the management fee is now consistently taken after losses have been realized.

6.5 Low-level STATICCALL Results in Reading Dirty Memory

Correctness Medium Version 1 Code Corrected

CS-MORPHO-VLT2-003

In `accrueInterestView()`, a low-level STATICCALL to the VIC is performed, without checking for code existence at that address. The subsequent check of the success flag is insufficient, because a STATICCALL to an address without code will return true.
The consequence is that the `output`, supposedly parsed from the returned `data` through a raw `mload(add(data,32))`, is actually read from dirty memory, because `data` will be an empty array. More specifically, `data` will point at location `0x60` (Solidity's zero-area) in memory, therefore `output` is read from location `0x80`.
This memory word is not just dirty, but can actually be under the attacker's control, in some call paths. For example, the function `forceDeallocate()` will first of all abi-decode its arguments, among which `address[] memory adapters`. This first abi-decode will write the length of the `adapters` array into the memory word `0x80`. The attacker can supply a long list of duplicated adapters, with 0-filled `assets`, so as to write any number he wants into that memory word, which will eventually be interpreted as the interest-per-second.

Code corrected:

The low-level STATICCALL has been replaced with a standard solidity call to the VIC, which will automatically revert if there is no code at the VIC address.

6.6 Interest Not Accrued When Set to Zero

Design Low Version 2 Specification Changed

CS-MORPHO-VLT2-018

When `zeroInterestPerSecond()` is called on the `ManualVic`, the interest is not accrued in the vault as it is done in `setInterestPerSecondAndDeadline()`. As an effect, the interest that was due to the vault before `zeroInterestPerSecond()` was called is never distributed.

Specification changed:

In Version 3, the codebase was refactored and no longer uses a VIC contract.

6.7 Interest Cannot Always Be Distributed

Design Low Version 1 Code Corrected

CS-MORPHO-VLT2-013

In cases where the underlying asset has a significant value per unit and low decimals, it can happen that the interest cannot be distributed, as it would be rounded down to `0` when taken per second.

Consider the following example:

- The asset has 8 decimals, e.g. WBTC
- `_totalAssets` = 1 WBTC = 1×10^8, hence the vault holds significant value
- `totalSupply` = 1 share = 1×10^8
- The Vault has existed for a while, made some gains, resulting in a 3% interest.
- Hence, `interestPerSecond` = 0.95 wei, rounding down to 0 wei.
- If called each block, `interest = interestPerSecond * elapsed` = $0 times 12$ = 0.
- Hence, if called every block, this Vault cannot accrue interest, even though it holds significant value.

For such vaults, the interest can either not be continuously distributed or not distributed at all.

Code corrected:

In Version 3 the VIC was removed from the codebase. The interest is computed directly from the values returned by the adapters and is not distributed per second.

6.8 Raw Decoding of STATICCALL's Return Data Does Not Check Length

Correctness Low Version 1 Specification Changed

CS-MORPHO-VLT2-004

In `accrueInterestView()`, a low-level STATICCALL to the VIC is performed, and the raw return data is stored in the variable `bytes memory data`. In order to be able to tolerate malfunctioning VICs that return data not decodable to a `uint256`, the array `data` is not parsed with `abi.decode()` (which can revert) but is instead parsed with a raw `mload(add(data,32))`.
However, no check is made that the length of `data` is equal to 32, which is a necessary condition for `data` to be the encoding of a `uint256`.

Specification changed:

In Version 2 the low-level STATICCALL has been replaced with a standard solidity call to the VIC. As a result, the call will now revert if the VIC does not return a `uint256` type.

In Version 3 the VIC was completely removed.

6.9 Sentinels Can Increase Interest Rate in ManualVic

Trust Low Version 1 Specification Changed

CS-MORPHO-VLT2-017

The sentinel role is meant to be used in case of emergency to quickly reduce the interest rate reported by a `ManualVic`. However, the sentinels can also increase the interest rate (up to the configured

maximum). This introduces an unnecessary risk, as the sentinels are trusted not to incorrectly increase the interest rate.

Specification changed:

As of Version 2 the sentinels can no longer increase the interest rate in the VIC. They can only set it to zero, as well as set the maximum to zero.

In Version 3 the VIC was removed from the codebase.

6.10 `totalAssets()` Is Out-Of-Sync

Design Low Version 1 Specification Changed

CS-MORPHO-VLT2-005

The value returned by `totalAssets()` does not return the actual value managed by the vault. The value does not include the accrued interest nor any potential unrealized loss.

Specification changed:

As of Version 2 the `totalAssets` function includes the accrued interest in its return value. It still only accounts for losses once someone has called the `realizeLoss()` function. There is now an incentive paid to the user who calls `realizeLoss()`. As a result, it can be expected that the losses will be realized shortly after they occur.

In Version 3 the `realizeLoss()` function has been removed. Now `totalAssets()` always returns the actual value managed by the vault, including any interest and losses.

6.11 Event Emitted Even When State Is Not Updated

Informational Version 2 Specification Changed

CS-MORPHO-VLT2-015

Events should be emitted on each important storage update of a smart contract in order to allow external observers to track important events in the contract's life. Emitting an event when a value is replaced by itself or when no other important update was made can be avoided as no new information is gained, and also incurs an unnecessary gas cost.

The function `VaultV2.realizeLoss()` emits an event even when `loss=0`.

Specification changed:

The `realizeLoss()` function was removed in Version 3 of the codebase.

6.12 Small Losses Can Be Offset by Interest

Informational | Version 2 | Specification Changed

CS-MORPHO-VLT2-014

When there is a loss in an adapter, it can be realized by calling the `realizeLoss()` function. However, if the loss is small, it may not be worth the gas cost to call this function. A frequent occurrence of small losses are rounding errors, where the shares received by the vault are rounded down by 1 wei, resulting in a tiny loss.

If `realizeLoss()` is not called, the loss will be offset over time by interest. For example, in `MorphoVaultV1Adapter`, the loss is calculated as:

```
uint256 loss = allocation() - IERC4626(morphoVaultV1).previewRedeem(shares);
```

As the `VaultV1` earns interest, the loss will be reduced, until it becomes zero.

Loss being covered this way will result in a different state than if it had been realized. Specifically:

- The `totalAssets` of `VaultV2` will not be reduced by the loss.
- There will be no performance fee taken on the interest that covered the loss.
- There will be no incentive shares minted as reward for `realizeLoss()`.

The core difference between realized losses and interest is that both are applied to the `allocation()` of the adapter, but only realized losses are applied to the `totalAssets`. Interest is not immediately added to the `totalAssets`. As a result, losses being offset by interest result in a different state compared to losses being realized.

Specification changed:

In Version 3 of the codebase, the `realizeLoss()` function was removed. The interest and losses are now realized automatically.

6.13 Buggy VIC Can Burn All Gas in STATICCALL Context

Informational | Version 1 | Code Corrected

CS-MORPHO-VLT2-006

In `accrueInterestView()`, a low-level STATICCALL to the VIC is performed, to query the current interest rate. If a malfunctioning VIC is utilized, that "inadvertently" performs a state-changing operation (like a token payment to an oracle, for example) in the STATICCALL context, the call fails and burns all the forwarded gas. By the 63/64 rule, the transaction has 1/64 of the original gas to finish execution. This implies that virtually all operations will become very expensive, because most of them comprise a call to `accrueInterestView()`.
Note that the `ManualVic` in scope for this review does not have this vulnerability, but this should be considered in new VIC implementations.

Code corrected:

In Version 2 the static call was replaced by a normal Solidity call.

In Version 3 the VIC was removed from the codebase.

6.14 No NatSpec for Functions

Informational Version 1 Code Corrected

CS-MORPHO-VLT2-010

The contract functions currently do not include any NatSpec comments to document functions, parameters, return values, events, or contract metadata.

NatSpec comments can provide clear and concise documentation for users and developers interacting with the smart contracts. They can also help maintainability of the codebase. As such, it is considered best practice to include NatSpec comments.

Code corrected:

The NatSpec of `VaultV2` have been improved.

6.15 VIC Must Not Revert

Informational Version 1 Specification Changed

CS-MORPHO-VLT2-021

As of Version 2, the `interestPerSecond()` function in the VIC contract must not revert. If it does, it will be impossible to withdraw from the vault, until the VIC is replaced or stops reverting.

This increases the trust required in the VIC contract, which can be changed by the curator (with a timelock).

Users should be extra cautious in case one of the following conditions is met:

- The timelock of `setVic()` is short.
- A new VIC is set.
- The VIC has privileged functions that can change the behavior of the contract.
- The VIC is deployed using an upgradeable proxy.
- The VIC can revert depending on the circumstances (e.g. depending on who the initiator of the transaction is).

The above conditions can lead to a situation where a vault with a previously working VIC can start reverting. In the worst case, a malicious curator could intentionally cause a VIC to revert and hold the users' funds for ransom, threatening to unfreeze the funds only if the ransom is paid.

Specification changed:

As of Version 3 the VIC was removed from the codebase.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Gas Optimizations

Informational **Version 1**

CS-MORPHO-VLT2-007

The following is a list of possible gas optimizations that could be applied to the codebase:

Version 1

1. The `DOMAIN_SEPARATOR` can be cached to avoid recomputing the hash every time, but must still be recomputed if the chain ID changes
2. In the two `skim()` functions, `msg.sender` can be used in place of `skimRecipient` in the transfer. The `skimRecipient` will trigger a hot SLOAD (100 gas) and the `msg.sender` will trigger a `CALLER` opcode (2 gas).
3. As `totalSupply >= balanceOf(account)` for arbitrary accounts holds, the following computations can be in an `unchecked` block:
    - `balanceOf[to] += shares` in `createShares()`
    - `totalSupply -= shares` in `deleteShares()`
4. During a transfer, as the amount of shares sent is the same as the amount of shares received and no new shares are created or deleted, the following computations can be in an `unchecked` block:
    - `balanceOf[to] += shares` in `transfer()`
    - `balanceOf[to] += shares` in `transferFrom()`
5. In `transferFrom()`, if the allowance must be decreased, the subtraction `_allowance - shares` is computed twice and could be cached instead.

Version 3

1. The length of the `adapters` array can be cached in `VaultV2.accrueInterestView()` to avoid redundant SLOADs
2. The length of the `marketParamsList` array can be cached in `MorphoMarketV1Adapter.realAssets()` to avoid redundant SLOADs

7.2 Malicious IRM Can Re-Enter and Multiply Reported Interest and Loss

Informational **Version 1** **Risk Accepted**

CS-MORPHO-VLT2-008

Both the `MorphoBlueAdapter` and the `MetaMorphoAdapter` include a checkpointing logic that does not strictly follow the CEI pattern: the new asset checkpoint value is only written after the external call into Morpho (or MetaMorpho, thus eventually into Morpho). This logic is therefore theoretically vulnerable to reentrancy attacks.

In all 4 call paths (`Morpho.supply()`, `Morpho.withdraw()`, `MetaMorpho.deposit()`, `MetaMorpho.withdraw()`), one of the earliest actions that happens before any tokens get transferred, is an internal call to `Morpho._accrueInterest()`, which calls the external non-view function `IIrm.borrowRate()`. If a malicious IRM is configured for this market (or if the IRM calls an untrusted contract), it can reenter the VaultV2 via `forceDeallocate()`: the gauged loss at the beginning of the `deallocate()` function will again be the same, because the checkpoint has not been updated yet, and the view function to measure the invested assets will also return the same value, because no tokens have been transferred yet.

In Version 2, the loss realization cannot be targeted by this, thanks to the order of the calls where the underlying system is called before querying the allocation. But the interest could be inflated because the value of the allocation will be cached before the reentrancy, and therefore will not be updated when the call unwinds, allowing to account for the difference `currentValue - allocation` multiple times.

In Version 3, the order of operations is such that the allocation is always read after interacting with the underlying system and is thus always up-to-date, even in the context of a reentrancy. The issue should be mitigated for `MorphoMarketV1Adapter`, but might still be present for `MorphoVaultV1Adapter` as the Vaults V1 are potentially vulnerable.

Risk accepted:

Morpho Labs has responded with the following statement:

```
Risk accepted. There is a similar potential vulnerability
in MetaMorpho as well if the IRM listing is permissionless,
so we don't plan to allow it.
```

7.3 Missing Events and Custom Errors

Informational | Version 1 | Code Partially Corrected

CS-MORPHO-VLT2-009

Following is a non-exhaustive list of places in the code that revert with Solidity's native underflow error, instead of a more explicative custom one:

1. In `transfer()` and `transferFrom()`, when subtracting the shares amount from the `balanceOf` mapping
2. In `transferFrom()` and `exit()`, when subtracting the shares amount from the allowance.

Also, in `exit()`, if the idle assets are insufficient, but no `liquidityAdapter` is set, the call will fail when the vault tries to transfer out more than its balance.

The following places in the code are missing an event that could help off-chain tracking of the system progress:

1. In the `timelocked` function, it could be useful to emit an event indicating that the call was executed and is not pending anymore. This could be useful to track pending and executed calls submitted through `submit()`.

Code partially corrected:

An event was added to the `timelocked` function.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Adapters Deployed by Factory

Note Version 1

The `MorphoMarketV1Adapter` and `MorphoVaultV1Adapter` can be deployed by their dedicated factories. Users must be aware that the deployment of such adapters is permissionless and therefore adapters deployed through their factories are not guaranteed to be well configured and must be carefully checked.

8.2 Allowing an Empty VaultV1 Can Lead to Inflation Attack

Note Version 1

The share price of `VaultV1` (MetaMorpho) can be inflated by depositing assets into Morpho Blue `onBehalf` of the `VaultV1`. If the `VaultV1` is initially empty, the inflation can be large enough to drain a `VaultV2` that uses the `VaultV1` as liquidity market.

Prerequisites:

- There is a `MorphoVaultV1Adapter` that uses a vault with a very low number of shares.
- There are funds to steal in other adapters or idle liquidity.
- The low share vault is the liquidity market OR the attacker is the allocator.

Scenario 1: This example assumes there are no `VaultV1` shares except for `1` virtual share.

1. Deposit `1` asset to the VaultV1. Now: `1` assets, `2` shares.
2. Deposit `1e18` assets into the Morpho market onBehalf of the `VaultV1`. This inflates the `VaultV1` share price. `50%` of this is a cost. Now: `1e18+1` assets,``2`` shares.
3. Deposit `0.5e18` assets to `VaultV2`. `VaultV2` deposits to `VaultV1`. The returned shares round to zero. This further inflates the share price. Now: `1.5e18+1``assets, ``2` shares.
4. Deposit `0.75e18` assets to VaultV2. Rounds to zero. Repeat, increasing the amount `1.5x` each round (scales exponentially), until the allocation of the `VaultV1Adapter` is equal to the initial TVL of the vault.
5. Call `forceDeallocate()` to free the other assets into idle liquidity. This costs `2%` of the attacker's shares (those `2%` also go to idle liquidity).
6. Attacker withdraws all their shares from `VaultV2`. As loss has not been realized yet, the attacker gets everything back (from idle liquidity).
7. Withdraw `1` share from `VaultV1`. This share is now worth `50%` of the initial TVL of VaultV2.
8. The vault loses `100%` of TVL. Attacker gets `49%` of it as profit.

The cost is `2%` for forceDeallocate, plus a negligible amount to inflate (step 2.). The looped steps (3./4.) are free. The inflation cost is negligible, as it was assumed there is no initial liquidity in the `VaultV1`. If there is more liquidity, step 2. will cost a lot more.

Scenario 2: This Scenario assumes there are `1e18+1` initial shares and `1e18` assets in the VaultV1.

1. Deposit `1e18` assets to the `VaultV1`. Now: `2e18` assets, `2e18+1` shares.
2. Deposit `1000000e18` assets into the Morpho market onBehalf of the `VaultV1`. This inflates the VaultV1 share price. `50%` of this is a cost. Now: `1000002e18` assets, `2e18+1` shares.
3. Deposit `500000` assets to `VaultV2`. `VaultV2` deposits to `VaultV1`. The returned shares round to zero. This further inflates the share price. Now: `1000002e18+500000` assets, `2e18+1` shares.
4. The shares are not inflated enough. We can still only deposit `500000` assets to `VaultV2` while rounding to zero. The exponential scaling does not work anymore.

Conclusion: It is critical for curators to ensure that there is a large number, with a significant value, of "dead" shares in every `VaultV1` that is used by a `MorphoVaultV1Adapter`. The same check should also be performed for all Morpho Blue markets used by the integrated vaults, as well as all Morpho Blue Markets used in a `MorphoMarketV1Adapter`.

For notes on how to check a market for safety, see How to Check Market Initial Liquidity for Safety.

8.3 Future Adapters

Note **Version 1**

The correctness of the adapters is critical to the security and solvency of the `VaultV2`. When developing new adapters, the following should hold:

- When returning IDs, the adapter must return at least one ID that identifies the most fine-grained representation of the underlying position that is being adjusted.
- This ID must be unique across all IDs of all adapters.
- This ID's allocation will be the lowest amount among all IDs returned for that position.
- For example, in `MorphoVaultV1Adapter`, this is the `adapterId`. In `MorphoMarketV1Adapter` it is `keccak256(abi.encode("this/marketParams", address(this), marketParams))`.

The following must hold:

- `ids()` must always return the same values for the same input.
- IDs returned by `allocate()/deallocate()` must always match the IDs returned by `ids()` for the same input.
- After a call to deallocate, the vault must have an approval to transfer at least `assets` from the adapter.
- It must be possible to make deallocate possible (for in-kind redemptions).
- It must be enforced that only the vault can call `allocate()/deallocate()`.
- Markets must be entered/exited only in `allocate()/deallocate()`.

Other assumptions to have in mind while developing new adapters:

- The adapter must not trust the data it receives on `deallocate()` as it can come from user-provided data through `forceDeallocate()`.
- The adapter or the integrated protocol must not have any arbitrary callbacks allowing a reentrancy.

8.4 How to Check Market Initial Liquidity for Safety

Note **Version 1**

As described in Allowing an empty VaultV1 can lead to Inflation Attack, it is critical for all underlying protocols used in adapters (Metamorpho Vaults, their allowed Markets, and Morpho Blue Markets used directly) to have sufficiently high "dead liquidity" to prevent a share price inflation attack.

Curators or users can gain confidence that an underlying vault or market is resistant to such attacks by verifying the following:

- Significant initial liquidity: There should be at least `$1` worth of shares minted and made irretrievable.
- Initial liquidity minted without inflation: The initial liquidity should be minted without inflation, meaning that the above value check must be done on the value of received shares, not just the asset deposit amount. If the number of shares minted by the initial deposit is not checked, it could be frontrun by an inflation attack. A contract could be used to check the minted shares and revert if there are fewer shares minted than expected.
- Irretrievable initial liquidity: The initial liquidity should be irretrievable, meaning that it cannot be withdrawn. This can be done by depositing it on behalf of an immutable contract with no withdrawal function, or by depositing it on behalf of a "dead address" that is known to be impossible to access, such as the address of a precompile.

Note that these checks are not exhaustive, as there are many factors influencing the ultimate safety, such as the `decimals` and price of the underlying tokens. E.g. a known exception is WBTC, where `$1` of dead shares may not be enough.

However, if one of these checks fails, it is a strong indication that protection is insufficient.

8.5 Normal Withdrawals Can Be Blocked by the Allocator

Note **Version 2**

In `VaultV2`, withdrawals deallocate from the liquidity market in order to service withdrawals if the idle liquidity is not sufficient. However, the allocator can set a liquidity market that is not whitelisted as `isAdapter`, which will force users to use `forceDeallocate()`.

Consider the following scenario where the allocator is acting maliciously:

1. Allocator sets a correct liquidity market and data.
2. Liquidity market is whitelisted with `setIsAdapter()`.
3. Users enter the vault and liquidity enters the liquidity market.
4. Allocator calls `setLiquidityAdapterAndData()` with a new non-whitelisted adapter address (this does not require a timelock).
5. `deallocate()` will revert, as `isAdapter` is false for the new adapter.
6. Users must use `forceDeallocate()` to withdraw, which has a penalty cost.

Deallocations reverting will temporarily force users to pay a penalty to withdraw. This can last until a new allocator is set by the curator, which requires waiting for a timelock to pass.

8.6 Notable ERC4626 Function Special Behaviors

Note Version 2

The `VaultV2` is, as of Version 2, compliant with EIP-4626 (https://eips.ethereum.org/EIPS/eip-4626). While compliant, the behavior of the following functions should be noted:

- The view functions `maxDeposit()/maxMint()/maxWithdraw()/maxRedeem` always return `0`. This is in line with the standard, which allows to underestimate the amount.

8.7 Solvency of the Vault Depends on the Solvency of Integrated Protocols

Note Version 1

If any of the underlying protocols that integrate with the vault become insolvent, the vault may also become insolvent if the deallocation is too slow or if it is impossible to redeem all its shares.

This is particularly relevant for the integration with `MetaMorphoV1.1`. If the underlying vault incurs a loss, it is impossible to realize this loss. This will trigger a bank run on MetaMorpho. The loss for the `VaultV2` can be either `0%` or `100%` of the allocated funds, depending on how quickly the funds are deallocated from the insolvent `MetaMorphoV1.1`. If there is a loss, it will trigger a bank run on `VaultV2`.

8.8 Vault Shares May Be Shorted

Note Version 1

With a procedure similar to what is described in Users can escape losses, anyone could short sell the vault shares, if they see a future loss coming, provided they manage to find a loan provider for those shares. The vault will take the loss because it will effectively act as the counterparty to the short.
Note that this cannot be done atomically using flashloans: redeeming and depositing in the same transaction always has the same price. This is due to the special behavior of the `accrueInterestView()` function when called multiple times in the same transaction.