

May 29, 2025

Vault V2

Smart Contract Security Assessment



Contents

About Zellic	4
<hr/>	
1. Overview	4
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5
<hr/>	
2. Introduction	6
2.1. About Vault V2	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	10
2.5. Project Timeline	11
<hr/>	
3. Detailed Findings	11
3.1. First deposit attack as a consequence of interest accrual	12
3.2. Incorrect max interest rate per second	13
3.3. Signature malleability in permit function	14
3.4. Force Deallocate penalty evasion	16
3.5. Users may lose interest when the accrueInterest function is not called in a timely manner	18

4.	Discussion	21
4.1.	Vault's NatSpec comment is inconsistent with the implementation	22
4.2.	Vault must be pinged often	24
<hr data-bbox="488 525 1567 529"/>		
5.	Threat Model	25
5.1.	Contract: MetaMorphoAdapter.sol	26
5.2.	Contract: MorphoBlueAdapter.sol	29
5.3.	Contract: VaultV2.sol	32
<hr data-bbox="488 846 1567 850"/>		
6.	Assessment Results	36
6.1.	Disclaimer	37

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Morpho from May 22th to May 29th, 2025. During this engagement, Zellic reviewed Vault V2's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any scenarios in which permissioned roles can deceive investors or users and steal funds without the timelock mechanism providing a warning?
 - Are allocation and deallocation caps properly enforced under all conditions?
 - Can flash loans be used to temporarily manipulate vaults in a way that allows an attacker to profit?
-

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

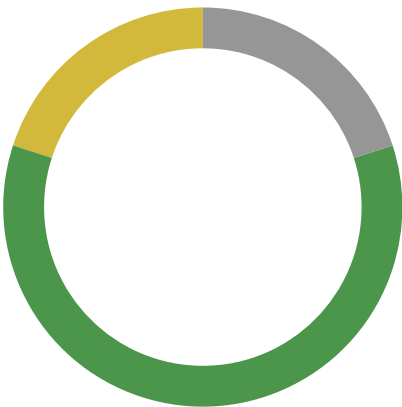
1.4. Results

During our assessment on the scoped Vault V2 contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, three were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Morpho in the Discussion section ([4. 7](#)).

Breakdown of Finding Impacts

Impact Level	Count
<div>Critical</div>	0
<div>High</div>	0
<div>Medium</div>	1
<div>Low</div>	3
<div>Informational</div>	1



2. Introduction

2.1. About Vault V2

Morpho contributed the following description of Vault V2:

Morpho Vault V2 enables anyone to create vaults that allocate assets to any protocols, including but not limited to Morpho Market V1, Morpho Market V2, and MetaMorpho Vaults. Depositors of Morpho Vault V2 earn from the underlying protocols without having to actively manage the risk of their position. Management of deposited assets is the responsibility of a set of different roles (owner, curator and allocators). The active management of invested positions involve enabling and allocating liquidity to protocols.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. 7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Vault V2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	Primary Review Period
Repository	https://github.com/morpho-org/vaults-v2 ↗
Version	77aa7c5baa823697ed7a35fa0df2fd0f3617be00
Programs	VaultV2.sol VaultV2Factory.sol adapters/*.sol imports/*.sol interfaces/*.sol libraries/*.sol vic/*.sol

Target	Secondary Review Period
Repository	https://github.com/morpho-org/vaults-v2 ↗
Version	Only changes between 77aa7c5...5938a92
Programs	VaultV2.sol VaultV2Factory.sol adapters/*.sol imports/*.sol interfaces/*.sol libraries/*.sol vic/ManualVic.sol vic/ManualVicFactory.sol vic/interfaces/IManualVic.sol vic/interfaces/IManualVicFactory.sol

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-weeks. The assessment was conducted by two consultants over the course of one calendar week.

Contact Information

The following project managers were associated with the engagement:

 **Jacob Goreski**
Engagement Manager
jacob@zellic.io ↗

 **Chad McDonald**
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

 **Kritsada Dechawattana**
Engineer
kritsada@zellic.io ↗

 **Ayaz Mammadov**
Engineer
ayaz@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 22, 2025	Kick-off call
---------------------	---------------

May 22, 2025	Start of primary review period
---------------------	--------------------------------

May 29, 2025	End of primary review period
---------------------	------------------------------

July 14, 2025	Start of secondary review period
----------------------	----------------------------------

July 17, 2025	End of secondary review period
----------------------	--------------------------------

3. Detailed Findings

3.1. First deposit attack as a consequence of interest accrual

Target	VaultV2		
Category	Business Logic	Severity	Medium
Likelihood	Low	Impact	Medium

Description

Share inflation should not be possible with the morpho vault design, in order to prevent malicious users from front-running a victim's deposit transaction to steal their assets. This aligns with morpho vault's current design as it keeps track of the total number of assets internally and does not allow donation.

However, a malicious attacker could exploit this by creating a normal vault, allowing it to accrue interest, then withdrawing all but one share. By intentionally triggering truncation, they can inflate the value of the remaining share. When a user attempts to enter the vault, the attacker can withdraw their remaining shares, leaving only the single, inflated share behind.

Impact

A user who wishes to enter a vault, might get frontrun and lose their funds to a truncation issue

Recommendations

Implement some sort of slippage check, to ensure that the user gets the amount of shares they expected otherwise revert.

Remediation

The team acknowledged the issue in commit [40767bf9](#) with a code comment, that the first deposit attack is still possible given certain circumstances and stated that the vaults would be required to be seeded with liquidity, though potentially further changes are required to make the seed deposit unremovable.

3.2. Incorrect max interest rate per second

Target	ConstantsLib.sol		
Category	Coding Mistakes	Severity	Low
Likelihood	High	Impact	Low

Description

The max interest rate per second is supposed to be equivalent to an interest gain of 200% APR. However, the max rate per second described in WADs is equivalent to 3 WADs (3e18).

```
uint256 constant MAX_RATE_PER_SECOND = (1e18 + 200 * 1e16)
/ uint256(365 days); // 200% APR
```

This would be equivalent to 200% APR if this value were assumed to be multiplicative, as in it is applied onto a principal as it would represent the principal plus 200% of the principal.

This is not the case as the interest rate is additive to the principal; as such, this is equivalent to $\text{principal} + (\text{principal} * \text{MAX_RATE_PER_SECONDS})$, which in this case would 4x the original principal, equivalent to 400% APR.

Impact

Vaults could have interest rates that are higher than intended, which could result in problematic scenarios down the line, such as too much interest being accrued but not being backed by actual yield, or it could be used to phish users by making the vault look temporarily extra attractive.

Recommendations

Adjust this number to be equivalent to the actual intended APR (1e18 or 1 WAD).

Remediation

This issue has been acknowledged by Morpho. This issue was remediated in commit [eaa69c5b](#) by changing the max interest rate per second to match 200% APR.

3.3. Signature malleability in permit function

Target	VaultV2.sol		
Category	Coding Mistakes	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The function `permit` is responsible for allowing ERC-20 permits to be used and does not account for signature malleability. As a result, every signature has a valid alternative.

```
function permit(address _owner, address spender, uint256 shares,
    uint256 deadline, uint8 v, bytes32 r, bytes32 s)
    external
{
    require(deadline >= block.timestamp, ErrorsLib.PermitDeadlineExpired());

    uint256 nonce = nonces[_owner]++;
    bytes32 hashStruct = keccak256(abi.encode(PERMIT_TYPEHASH, _owner,
        spender, shares, nonce, deadline));
    bytes32 digest = keccak256(abi.encodePacked("\x19\x01",
        DOMAIN_SEPARATOR(), hashStruct));
    address recoveredAddress = ecrecover(digest, v, r, s);
    require(recoveredAddress != address(0) && recoveredAddress == _owner,
        ErrorsLib.InvalidSigner());

    allowance[_owner][spender] = shares;
    emit EventsLib.Approval(_owner, spender, shares);
    emit EventsLib.Permit(_owner, spender, shares, nonce, deadline);
}
```

Impact

In the context of this `permit` function, the use of an incrementing nonce prevents replay attacks by ensuring that each signature can only be used once. Without it, an alternative valid signature could be reused to execute the permit function multiple times, despite only being intended for a single use.

Recommendations

Implement the [necessary checks](#) to ensure that only one signature may be used similar to industry standard implementations such as OpenZeppelin which uses signature malleability checks in addition to a nonce.

Remediation

This issue has been acknowledged by Morpho.

Morpho provided the following response to this finding:

The use of a nonce preventing the malleability is acceptable for us.

3.4. Force Deallocate penalty evasion

Target	VaultV2		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

```
function forceDeallocate(address[] memory adapters, bytes[] memory data,
    uint256[] memory assets, address onBehalf)
    external
    returns (uint256)
{
    require(adapters.length == data.length && adapters.length ==
        assets.length, ErrorsLib.InvalidInputLength());
    uint256 penaltyAssets;
    for (uint256 i; i < adapters.length; i++) {
        this.deallocate(adapters[i], data[i], assets[i]);
        penaltyAssets
        += assets[i].mulDivDown(forceDeallocatePenalty[adapters[i]], WAD);
    }
    ...
}
```

Depending on the amount of assets and the amount of forceDeallocatePenalty, penalties can be avoided by supplying a small enough amount such that the division with a WAD would truncate to 0. This could negatively affect smaller decimal assets such as USDC.

Impact

imagine a forceDeallocatePenalty of 0.2%, a user would be able to withdraw 500 dust (1/20 of a cent) of USDC per loop without paying the 0.2% penalty. Though this has almost no impact on 18 digit decimals.

Recommendations

Round up the penalty instead of rounding down.

Remediation

This issue was remediated in commit [51ebe7fd](#) by rounding up the penalty.

3.5. Users may lose interest when the accrueInterest function is not called in a timely manner

Target	VaultV2, ManualVic		
Category	Business Logic	Severity	Low
Likelihood	Low	Impact	Low

Description

The VaultV2 contract distributes interest to users over time until a predefined deadline, which is managed by the ManualVic contract. However, if the VaultV2::accrueInterest function (or any function that internally invokes it) is not called before the deadline, users may lose a portion of the interest they are entitled to regardless of how long they have been deposited prior to that point.

The root cause lies in how ManualVic::interestPerSecond is handled. Specifically, VaultV2::accrueInterestView retrieves the interest rate from IVic(vic).interestPerSecond(). This function, implemented in ManualVic, simply returns the storedInterestPerSecond if the current time is before the deadline; otherwise, it returns zero:

```
/// @dev Returns the interest per second.
function interestPerSecond(uint256, uint256) external view returns (uint256) {
    return block.timestamp <= deadline ? storedInterestPerSecond : 0;
}
```

Thus, when accrueInterest is called after the deadline, the interest rate is zero, and no retroactive interest is calculated for the period between the last accrual and the deadline:

```
function accrueInterestView() public view returns (uint256, uint256,
uint256) {
    uint256 elapsed = block.timestamp - lastUpdate;
    if (elapsed == 0) return (_totalAssets, 0, 0);

    @>    uint256 tentativeInterestPerSecond = vic != address(0) ?
IVic(vic).interestPerSecond(_totalAssets, elapsed) : 0;

    uint256 interestPerSecond = tentativeInterestPerSecond
        <= uint256(_totalAssets).mulDivDown(MAX_RATE_PER_SECOND, WAD) ?
tentativeInterestPerSecond : 0;
    uint256 interest = interestPerSecond * elapsed;
    uint256 newTotalAssets = _totalAssets + interest;
```

```
[...]
}
```

Proof of Concept

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity 0.8.28;

import "./BaseTest.sol";

contract InterestDeadlineTest is BaseTest {
    using MathLib for uint256;

    address performanceFeeRecipient = makeAddr("performanceFeeRecipient");
    address managementFeeRecipient = makeAddr("managementFeeRecipient");

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

        vm.startPrank(curator);
        vault.submit(abi.encodeCall(IVaultV2.setPerformanceFeeRecipient,
            (performanceFeeRecipient)));
        vault.submit(abi.encodeCall(IVaultV2.setManagementFeeRecipient,
            (managementFeeRecipient)));
        vm.stopPrank();

        vault.setPerformanceFeeRecipient(performanceFeeRecipient);
        vault.setManagementFeeRecipient(managementFeeRecipient);

        deal(address(underlyingToken), address(this), type(uint256).max);
        underlyingToken.approve(address(vault), type(uint256).max);
    }

    function testAccrueInterestBeforeDeadline() public {
        uint256 deposit = 1e18;
        uint256 deadline = block.timestamp + 30 days;
        uint256 expectedInterest = 164383561642464000;

        vault.deposit(deposit, address(this));
        uint256 beforeAccrueInterest = vault.totalAssets();
        vm.prank(allocator);

        vic.setInterestPerSecondAndDeadline(deposit.mulDivDown(MAX_RATE_PER_SECOND,
            WAD), deadline);

        vm.warp(deadline - 15 days);
    }
}
```

```

        vault.accrueInterest();
        uint256 afterAccrueInterest = vault.totalAssets();
        uint256 actualInterest = afterAccrueInterest - beforeAccrueInterest;

        assertEq(actualInterest, expectedInterest / 2); // accrue 15 days of
interest
    }

    function testAccrueInterestAtDeadline() public {
        uint256 deposit = 1e18;
        uint256 deadline = block.timestamp + 30 days;
        uint256 expectedInterest = 164383561642464000;

        vault.deposit(deposit, address(this));
        uint256 beforeAccrueInterest = vault.totalAssets();
        vm.prank(allocator);

        vic.setInterestPerSecondAndDeadline(deposit.mulDivDown(MAX_RATE_PER_SECOND,
WAD), deadline);

        vm.warp(deadline);
        vault.accrueInterest();
        uint256 afterAccrueInterest = vault.totalAssets();
        uint256 actualInterest = afterAccrueInterest - beforeAccrueInterest;

        assertEq(actualInterest, expectedInterest); // accrue 30 days of
interest
    }

    function testAccrueInterestAfterDeadline() public {
        uint256 deposit = 1e18;
        uint256 deadline = block.timestamp + 30 days;

        vault.deposit(deposit, address(this));
        uint256 beforeAccrueInterest = vault.totalAssets();
        vm.prank(allocator);

        vic.setInterestPerSecondAndDeadline(deposit.mulDivDown(MAX_RATE_PER_SECOND,
WAD), deadline);

        vm.warp(deadline + 1 days);
        vault.accrueInterest();
        uint256 afterAccrueInterest = vault.totalAssets();
        uint256 actualInterest = afterAccrueInterest - beforeAccrueInterest;

        assertEq(actualInterest, 0); // accrue zero interest instead of the 30
days of interest
    }

```

```
}  
}
```

Impact

Users may permanently lose interest earnings if the `accrueInterest` function is not called in timely, despite being eligible based on their deposit duration.

Recommendations

Implement some sort of interest accrual to cover in case that the `accrueInterest` function is not called in a timely manner. This ensures that the interest accrued between the last update and the deadline is accounted for even if the `accrueInterest` function is triggered after the deadline.

Remediation

This issue has been acknowledged by Morpho, and a fix was implemented in commit [0e180ad8](#).

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Vault's NatSpec comment is inconsistent with the implementation

The NatSpec comment in the VaultV2 contract at line 32 has a loose specification of adapters:

“they must have approved assets for the vault at the end of deallocate”.

However, in the actual implementation, it approves the maximum `uint256` value for Vault once in the adapter's constructor and lets Vault freely transfer in each operation. We recommend ensuring the comment is consistent with the implementation.

```
[...]
    /// @dev Loose specification of adapters:
    /// - They must enforce that only the vault can call allocate/deallocate.
    /// - They must enter/exit markets only in allocate/deallocate.
    /// - They must return the right ids on allocate/deallocate.
@>    /// - They must have approved `assets` for the vault at the end of
    deallocate.
    /// - They must make it possible to make deallocate possible (for in-kind
    redemptions).
    /// @dev Liquidity market:
    /// - `liquidityAdapter` is allocated to on deposit/mint, and deallocated
    from on withdraw/redeem if idle assets don't
    /// cover the withdraw.
    /// - The liquidity market is mostly useful on exit, so that exit liquidity
    is available in addition to the idle assets.
    /// But the same adapter/data is used for both entry and exit to have the
    property that in the general case looping
    /// supply-withdraw or withdraw-supply should not change the allocation.
    contract VaultV2 is IVaultV2 {

        [...]

        function deallocate(address adapter, bytes memory data,
            uint256 assets) external {

            [...]

@>            SafeERC20Lib.safeTransferFrom(asset, adapter, address(this),
            assets);
```

```
        emit EventsLib.Deallocate(msg.sender, adapter, assets, ids, loss);
    }

    [...]

}
```

```
contract MorphoBlueAdapter is IMorphoBlueAdapter {

    [...]

    constructor(address _parentVault, address _morpho) {
        morpho = _morpho;
        parentVault = _parentVault;
        SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _morpho,
            type(uint256).max);
    @> SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _parentVault,
        type(uint256).max);
    }

    [...]

}
```

```
contract MetaMorphoAdapter is IMetaMorphoAdapter {

    [...]

    constructor(address _parentVault, address _metaMorpho) {
        parentVault = _parentVault;
        metaMorpho = _metaMorpho;
    @> SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _parentVault,
        type(uint256).max);
        SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _metaMorpho,
            type(uint256).max);
    }

    [...]

}
```

Remediation

This was fixed in commit [832c53d98555ca4652d8397e2738e6f4431e80be](#) by changing the comments to be consistent with the implementation.

4.2. Vault must be pinged often

Interest accrual is implemented as a public function that can be called directly by users or triggered when other public functions that may affect interest accrual, such as withdraw, deposit, etc., are called.

Due to the implementation of the interest accrual, the vault must be pinged often to maintain fair and equal distribution of shares between the performance and management fee according to the distribution in the vault. This is mainly because the number of management fee assets is calculated as $(\text{newTotalAssets} * \text{elapsed} * \text{managementFee}) / \text{WAD}$, meaning that delayed accruals result in more management fee assets, and therefore more management fee shares.

While the differences are not huge, the vault must be pinged monthly at least to keep the distribution fair, otherwise a difference of ~3-5% can be noted.

Here are the share distributions we analyzed across different ping intervals.

```
yearly update
  a vault total assets: 5220752000
  a vault performance fee shares: 107992053
  a vault management fee shares: 212833001

monthly update
  a vault total assets: 5220631040
  a vault performance fee shares: 111044394
  a vault management fee shares: 206722683

weekly update
  a vault total assets: 5220776192
  a vault performance fee shares: 111335864
  a vault management fee shares: 206451329

daily update
  a vault total assets: 5220752000
  a vault performance fee shares: 111379460
  a vault management fee shares: 206321870

hourly update
  a vault total assets: 5220752000
  a vault performance fee shares: 111384042
```



```
a vault management fee shares: 206296350  
  
update ever 5 mins  
a vault total assets: 5220752000  
a vault performance fee shares: 111335316  
a vault management fee shares: 206196124
```

Remediation

Users should treat the formula implemented in the code as the expected behavior. For this reason, and because the differences are not significant, this concern is acceptable while leaving the code as is.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Contract: MetaMorphoAdapter.sol

Function: `allocate(bytes data, uint256 assets)`

This function is responsible for taking funds allocated by the vault and accurately entering a position in the underlying MetaMorpho market while accounting for potential loss in the markets due to a multitude of things (such as bad debt, volatile markets, etc...)

Inputs

- `data`
 - **Control:** Full control.
 - **Constraints:** Must be empty.
 - **Impact:** Unused.
- `assets`
 - **Control:** Full control.
 - **Constraints:** Must not exceed the relative cap and absolute cap.
 - **Impact:** Amount of assets allocated from idle assets to the target market/protocol.

Branches and code coverage

Intended branches

- Only permissioned senders are allowed to call `allocate/deallocate`.
 - ☒ Test coverage
- First `allocate` — loss should be zero.
 - ☒ Test coverage
- `Allocate` supplies tokens to the underlying MetaMorpho vault.
 - ☒ Test coverage
- `Deallocate` should return zero loss on the first invocation.
 - ☒ Test coverage

- Deallocate should withdraw tokens from the underlying market.
 - ☑ Test coverage
- Update the `_totalAssets` after allocate when the loss occurs.
 - ☑ Test coverage
- Test a valid scenario with loss realization.
 - ☑ Test coverage

Function call analysis

- `IERC4626(metaMorpho).deposit(0, address(this))`
 - **What is controllable?** Nothing.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `uint256 loss = assetsInMetaMorpho.zeroFloorSub(IERC4626(metaMorpho).previewRedeem(IERC4626(metaMorpho).balanceOf(address(this))))`
 - **What is controllable?** Nothing.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned Loss may be controllable due to mechanics of the underlying morpho market, it may also be manipulated to include rounding errors.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IERC4626(metaMorpho).deposit(assets, address(this))`
 - **What is controllable?** `assets`.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `deallocate(bytes data, uint256 assets)`

This function allows Vault's allocators to deallocate the allocation from a given market/protocol to idle assets based on their strategy.

Inputs

- `data`

- **Control:** Full control.
 - **Constraints:** Must be empty.
 - **Impact:** Unused.
- assets
 - **Control:** Full control.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets to deallocate from the target market/protocol to idle assets.

Branches and code coverage

Intended branches

- Deallocate should return zero loss on the first invocation.
 - ☒ Test coverage
- Deallocate should withdraw tokens from the underlying market.
 - ☒ Test coverage
- Update the `_totalAssets` after allocating when the loss occurs.
 - ☒ Test coverage
- Test a valid scenario with loss realization.
 - ☒ Test coverage

Function call analysis

- `IERC4626(metaMorpho).deposit(0, address(this))`
 - **What is controllable?** Nothing.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `uint256 loss = assetsInMetaMorpho.zeroFloorSub(IERC4626(metaMorpho).previewRedeem(IERC4626(metaMorpho).balanceOf(address(this))))`
 - **What is controllable?** Nothing.
 - **If the return value is controllable, how is it used and how can it go wrong?** The returned Loss may be controllable due to mechanics of the underlying morpho market, it may also be manipulated to include rounding errors.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IERC4626(metaMorpho).withdraw(assets, address(this), address(this))`

- **What is controllable?** assets.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.2. Contract: MorphoBlueAdapter.sol

Function: `allocate(bytes data, uint256 assets)`

This function is responsible for taking funds allocated by the vault and accurately entering a position in the underlying Morpho-blue market while accounting for potential loss in the markets due to a multitude of things (such as bad debt, volatile markets, etc...)

Inputs

- `data`
 - **Control:** Full control.
 - **Constraints:** The decoded ID from `data` must be valid.
 - **Impact:** The market/protocol ID that will be allocated according to the specified assets.
- `assets`
 - **Control:** Full control.
 - **Constraints:** Must not exceed the relative cap and absolute cap.
 - **Impact:** Amount of assets allocated from idle assets to the target market/protocol.

Branches and code coverage

Intended branches

- Only permissioned senders are allowed to call `allocate/deallocate`.
 - ☒ Test coverage
- First `allocate` — loss should be zero.
 - ☒ Test coverage
- `Allocate` supplies tokens to the underlying Morpho-blue market.
 - ☒ Test coverage
- `Deallocate` should return zero loss on the first invocation.
 - ☒ Test coverage

- Deallocate should withdraw tokens from the underlying market.
 - ☑ Test coverage
- Update the `_totalAssets` after allocating when the loss occurs.
 - ☑ Test coverage
- Test a valid scenario with loss realization.
 - ☑ Test coverage

Function call analysis

- `IMorpho(morpho).accrueInterest(marketParams)`
 - **What is controllable?** `marketParams`.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `assetsInMarket[marketId].zeroFloorSub(IMorpho(morpho).expectedSupplyAssets(marketId, address(this)))`
 - **What is controllable?** `marketParams`.
 - **If the return value is controllable, how is it used and how can it go wrong?** If the loss returned is inaccurate from rounding down, the `_totalAssets` in the vault will be inaccurate as well.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `IMorpho(morpho).withdraw(marketParams, assets, 0, address(this), address(this))`
 - **What is controllable?** `marketParams` and `assets`.
 - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** Flash-loan-based shorting has been prevented.

Function: `deallocate(bytes data, uint256 assets)`

This function allows Vault's allocators to deallocate the allocation from a given market/protocol to idle assets based on their strategy.

Inputs

- data
 - **Control:** Full control.
 - **Constraints:** The decoded ID from data must be valid.
 - **Impact:** The market/protocol ID that will be deallocated according to the specified assets.
- assets
 - **Control:** Full control.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets deallocated from the target market/protocol to idle assets.

Branches and code coverage

Intended branches

- Deallocate should return zero loss on the first invocation.
 - ☒ Test coverage
- Deallocate should withdraw tokens from the underlying market.
 - ☒ Test coverage
- Update the `_totalAssets` after allocating when the loss occurs.
 - ☒ Test coverage
- Test a valid scenario with loss realization.
 - ☒ Test coverage

Function call analysis

- `IAdapter.deallocate`
 - **What is controllable?** `ids` and `loss`.
 - **If the return value is controllable, how is it used and how can it go wrong?** If the `loss` returned is inaccurate from rounding down, the `_totalAssets` will be inaccurate as well.
 - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

5.3. Contract: VaultV2.sol

Function: `allocate(address adapter, bytes data, uint256 assets)`

This function allows Vault's allocators to allocate the idle assets to a given market/protocol based on their strategy.

Inputs

- `adapter`
 - **Control:** Full control.
 - **Constraints:** Only the adapter that is set to be `true` on `isAdapter` is allowed.
 - **Impact:** The market/protocol in a given adapter will be allocated according to the specified assets.
- `data`
 - **Control:** Full control.
 - **Constraints:** The decoded ID from `data` must be valid.
 - **Impact:** The market/protocol ID will be allocated according to the specified assets.
- `assets`
 - **Control:** Full control.
 - **Constraints:** Must not exceed the relative cap and absolute cap.
 - **Impact:** Amount of assets allocated from idle assets to the target market/protocol.

Branches and code coverage

Intended branches

- Accumulate an interest before allocating.
 - ☒ Test coverage
- Prevents flash-loan-based shorting of Vault shares during loss realizations by setting the `enterBlocked` to be `true`.
 - ☒ Test coverage
- Underlying token balance of Vault should be zero when entirely allocating idle assets.
 - ☒ Test coverage
- Underlying token balance of Vault should be decreased when partially allocating idle assets.
 - ☐ Test coverage

- Underlying token balance of the adapter should increase from the assets amount.

☒ Test coverage

- Update the `_totalAssets` after allocating when the loss occurs.

☒ Test coverage

Negative behavior

- Caller is not the allocator or vault contract itself.

☒ Negative test

- The adapter input not already set to be `true`.

☒ Negative test

- Allocate with invalid ID.

☐ Negative test

- Does not pass absolute cap check `allocation[ids[i]] <= absoluteCap[ids[i]]`.

☒ Negative test

- Does not pass relative cap check `allocation[ids[i]] <= uint256(_totalAssets).mulDivDown(relativeCap[ids[i]], WAD)`.

☒ Negative test

Function call analysis

- `IAdapter.allocate`

- **What is controllable?** `ids` and `loss`.
- **If the return value is controllable, how is it used and how can it go wrong?** If the `loss` returned is inaccurate from rounding down, the `_totalAssets` will be inaccurate as well.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

Function: `deallocate(address adapter, bytes data, uint256 assets)`

This function allows Vault's allocators to deallocate the allocation from a given market/protocol to idle assets based on their strategy.

Inputs

- `adapter`
- **Control:** Full control.

- **Constraints:** Only the adapter that is set to be `true` on `isAdapter` is allowed.
 - **Impact:** The market/protocol in a given adapter will be deallocated according to the specified assets.
 - `data`
 - **Control:** Full control.
 - **Constraints:** The decoded ID from `data` must be valid.
 - **Impact:** The market/protocol ID will be deallocated according to the specified assets.
 - `assets`
 - **Control:** Full control.
 - **Constraints:** N/A.
 - **Impact:** Amount of assets deallocated from the target market/protocol to idle assets.

Branches and code coverage

Intended branches

- Prevents flash-loan-based shorting of Vault shares during loss realizations by setting the `enterBlocked` to be `true`.
 - ☒ Test coverage
- Underlying token balance of Vault should be increased from deallocating.
 - ☒ Test coverage
- Update the `_totalAssets` after deallocating when the loss occurs.
 - ☒ Test coverage

Negative behavior

- Caller is not the allocator, sentinel, or Vault contract itself.
 - ☒ Negative test
- The adapter input is not already set to be `true`.
 - ☒ Negative test
- Deallocate with invalid ID.
 - ☐ Negative test

Function call analysis

- `IAdapter.deallocate`

- **What is controllable?** `ids` and `loss`.
- **If the return value is controllable, how is it used and how can it go wrong?** If the `loss` returned is inaccurate from rounding down, the `_totalAssets` will be inaccurate as well.
- **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

Function: `forceDeallocate(address[] adapters, bytes[] data, uint256[] assets, address onBehalf)`

This function allows a depositor to force deallocate any markets/protocols with penalty of up to 2%.

A depositor can achieve in-kind redemptions by taking a flash loan and supplying it to the underlying market. This creates liquidity that the depositor can transfer to the idle market using this function. After withdrawing the funds and repaying the flash loan, the depositor will hold market shares instead of vault shares.

Inputs

- `adapters`
 - **Control:** Full control.
 - **Constraints:** Only the adapter that is set to be true on `isAdapter` is allowed.
 - **Impact:** The allocation in a market/protocol will be deallocated according to the specified assets value within the given list of assets.
- `data`
 - **Control:** Full control.
 - **Constraints:** All decoded IDs from `data` must be valid.
 - **Impact:** List of market/protocol IDs that will be deallocated according to the specified assets.
- `assets`
 - **Control:** Full control.
 - **Constraints:** N/A.
 - **Impact:** List of the amount of assets needed to deallocate from the target market/protocol to idle assets.
- `onBehalf`
 - **Control:** Full control.
 - **Constraints:** The caller must be either calling on their own behalf or have sufficient allowance.
 - **Impact:** The address to be charged the penalty fee.

Branches and code coverage

Intended branches

- Underlying token balance of Vault should be increased from deallocating.
 - ☒ Test coverage
- The shares of the `onBehalf` address should be charged according to the set penalty percentage.
 - ☒ Test coverage

Negative behavior

- The `adapters`, `data`, and `assets` lengths do not align.
 - ☒ Negative test
- One of the IDs from `data` is invalid.
 - ☐ Negative test

Function call analysis

- `VaultV2.deallocate`
 - **What is controllable?** `adapters[i]`, `data[i]`, and `assets[i]`.
 - **If the input value is controllable, how is it used and how can it go wrong?** N/A.
 - **What happens if it reverts, reenters, or does other unusual control flow?** If this reverts, the entire transaction would revert — no reentrancy scenarios.

6. Assessment Results

During our assessment on the scoped Vault V2 contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, three were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.