Morpho Blue

Security Review

Cantina Managed review by:

Saw-mon-and-Natalie, Lead Security Researcher

Jonah1005, Lead Security Researcher

StErMi, Security Researcher

April 22, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Directly* exploitable security vulnerabilities that need to be fixed.
High	Security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All high issues should be addressed.
Medium	Objective in nature but are not security vulnerabilities. Should be addressed unless there is a clear reason not to.
Low	Subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. When determining the severity one first needs to determine whether the finding is subjective or objective. All subjective findings are considered of Minor severity.

Next it is determined whether the finding can be regarded as a security vulnerability. Some findings might be objective improvements that need to be fixed, but do not impact the project's security overall (Medium).

Finally, objective findings of security vulnerabilities are classified as either critical or major. Critical findings should be directly vulnerable and have a high likelihood of being exploited. Major findings on the other hand may require specific conditions that need to be met before the vulnerability becomes exploitable.

2 Security Review Summary

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

It enables the deployment of minimal and isolated lending markets by specifying one loan asset, one collateral asset, a liquidation LTV (LLTV), and an oracle. The protocol is trustless and was designed to be more efficient and flexible than any other decentralized lending platform.

From Sep 28th - Oct 16th the Cantina team conducted a review of Morpho Blue on commit hash 11e69b...810c53. The team identified a total of **27** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 4
- Medium Risk: 1
- Low Risk: 4
- Gas Optimizations: 1
- Informational: 17

3 Findings

3.1 High Risk

3.1.1 Withdrawal or borrowing of loan tokens can be griefed

Severity: High Risk

Context: Morpho.sol, Morpho.sol#L451-L453

Description: With the current implementation of `Morpho` interest does not accrue within the same block and one is not disallowed to borrow and repay in the same block. These two properties open a griefing attack that:

1. Borrowing liquidity can be blocked.
2. Withdrawing liquidity by the suppliers can be blocked.

To perform such attacks the attacker would only need to:

1. Pay for transaction gas fees.
2. Have enough collateral to be able to borrow all the remaining liquidity.

The attack works as follows assume T_1 is the transaction the honest actor wants to submit to either borrow or withdraw liquidity, then the attacker frontruns T_1 with T_0 which is the transaction to remove/borrow all the liquidity by providing enough collateral and back runs with T_2 which is the transaction that would repay the loan and withdraws the collateral. Overall, the attacker would get all its collateral back but indeed would need to spend on gas and depending on the market state have enough liquidity to begin with.

Griefing the `borrow` end point can be tolerable as the honest actor would still have access to their tokens. But blocking a supplier to `withdraw` their loan tokens is one that is more important.

Proof of concept: Add the following test case to `WithdrawIntegrationTest`:

```
function testWithdrawBorrowSandwichAttack() public {
    address ATTACKER = _addrFromHashedString("Attacker");
    vm.startPrank(ATTACKER);
    loanToken.approve(address(morpho), type(uint256).max);
    collateralToken.approve(address(morpho), type(uint256).max);
    vm.stopPrank();

    uint256 amountSupplied = 100 ether;
    // minimum collateral needed to be supplied to borrow the whole supply asset and leave 0 liquidity
    uint256 amountCollateral = 125 ether;
    uint256 amountBorrowed = amountSupplied;

    loanToken.setBalance(SUPPLIER, amountSupplied);

    vm.prank(SUPPLIER);
    morpho.supply(marketParams, amountSupplied, 0, SUPPLIER, hex"");

    collateralToken.setBalance(ATTACKER, amountCollateral);
    collateralToken.setBalance(BORROWER, amountCollateral);

    vm.startPrank(ATTACKER);
    morpho.supplyCollateral(marketParams, amountCollateral, ATTACKER, hex"");
    morpho.borrow(marketParams, amountBorrowed, 0, ATTACKER, ATTACKER);
    vm.stopPrank();

    vm.startPrank(SUPPLIER);
    uint256 suppliedShares = morpho.supplyShares(marketParams.id(), SUPPLIER);
    vm.expectRevert(bytes(ErrorsLib.INSUFFICIENT_LIQUIDITY));
    morpho.withdraw(marketParams, 0, suppliedShares, SUPPLIER, RECEIVER);
    vm.stopPrank();

    vm.startPrank(BORROWER);
    morpho.supplyCollateral(marketParams, amountCollateral, BORROWER, hex"");
    vm.expectRevert(bytes(ErrorsLib.INSUFFICIENT_LIQUIDITY));
    morpho.borrow(marketParams, 1, 0, BORROWER, BORROWER); // just borrow 1 loan token
    vm.stopPrank();

    vm.startPrank(ATTACKER);
```

```
    morpho.repay(marketParams, amountBorrowed, 0, ATTACKER, hex"");
    morpho.withdrawCollateral(marketParams, amountCollateral, ATTACKER, ATTACKER);
    vm.stopPrank();
}
```

Recommendation: Perhaps borrowing and repaying in the same block (or even a small time interval) should be disincentived to prevent such attacks.

Morpho: We acknowledge this issue. While it's important to note, we don't think that it necessitates a fix. It's also present in AaveV2, V3, CompoundV2, V3 and afaik didn't cause issues. If one day it becomes a problem, users could push their transactions through something like MEV Blocker which prevents front-runs.

3.1.2 Consider adding additional sanity checks in `createMarket` to ensure that a market is created with valid parameters

Severity: High Risk

Context: Morpho.sol#L140-L151

Description: The current implementation of `createMarket` is not covering all the `marketParams` attributes with sanity checks. This could allow the creation of broken markets or markets that do not make sense.

Morpho should consider implementing the following sanity checks:

- `marketParams.loanToken != address(0) && loanToken != collateralToken`.
- `marketParams.collateralToken != address(0)` (the `collateralToken != loanToken` check is implicit because of the previous check).
- `marketParams.oracle != address(0)`.
- `marketParams.oracle.price()` returns a valid answer and does not revert.
- `marketParams.irm` is working as expected without reverting.

Recommendation: Consider implementing the suggested sanity check or clearly acknowledge them in the `IMorpho` natspec documentation, explaining which possible side effects could happen because those checks have not been performed.

Morpho We should do those checks because:

- We cannot cover everything, so why choose those specific scenarios ? There are many other checks that we could do.
- We already documented what are considered well-behaved markets, which should cover those checks.
- Creating a market is a complex task to do well, so we can expect that markets that are created in good faith are not doing those mistakes. Also, being able to freely create a market is a feature.
- There are some use cases that are restricted by those checks. Notably the case where `collateralToken = address(0)` represents a reserve of liquidity that as the same interface as Blue

3.1.3 User's funds will be stuck forever if an enabled `IRM` breaks

Severity: High Risk

Context: Morpho.sol#L456

Description: The `Morpho` protocol is a permissionless protocol that does not have any mechanism to recover funds or disable `IRM` and `LLTV` that have been enabled.

This approach allows Morpho to have a fully permissionless protocol, but has the drawback of enabling the possibility to lock users funds in market that have been already created and, more importantly, to allowing the creation of markets that will be for sure broken in the scenario where an `IRM` stop working (it will revert when `borrowRate` is called).

Let's assume that the enabled `IRM` is broken and when `IIrm(marketParams.irm).borrowRate` inside `_accrueInterest` is executed, the whole transaction reverts.

If the market is a "mature market" (`market[id].totalBorrowAssets > 0`) all the user-facing operation will revert. The only operation that can be executed (because it does not execute `_accrueInterest`) is `supplyCollateral` that will inject more collateral that won't be possible to withdraw.

The same situation can be reached on a freshly created market. Because `IRM` can't be disabled, even if the `IRM` is broken, no one can be stopped to execute `createMarket` by specifying the broken `IRM` as a parameter. Once the market has been created, anyone can start interacting with it. Let's see an example that will lead to locking the user's funds:

1) `Supplier1` supplies `100 ETH`, `IRM` is not triggered because `market[id].totalBorrowAssets == 0`.
2) `Borrower1` supplies `10_000 USDC` as collateral, `IRM` is not triggered because `market[id].totalBorrowAssets == 0`.
3) `Borrower1` borrows `1 ETH`, `IRM` is not triggered because `market[id].totalBorrowAssets == 0` (`_accrueInterest` is executed **before** that the market variables are updated).
4) At this point, any operations that call `_accrueInterest` will revert, preventing users to supply/withdraw/repay/liquidate and so on.

The only one that can be called is `supplyCollateral` that would inject into the market collateral tokens that can't be withdrawn (because `IRM` will revert).

Recommendation: If Morpho does want to keep the fully permissionless behavior of the protocol, they should at least carefully document and explain these possibilities to their users and integrators.

Otherwise, Morpho could introduce a new mechanism that will allow the owner to disable enabled `IRM`s and `LLTV`s and prevent at least the creation of new markets once those values have been disabled. The management of already created market that are using IRMs and LLTVs that have been disabled is a much more complex implementation that is out of the context of the issue itself.

Cantina: Morpho has improved the documentation about the consequences of a broken market: funds could get stuck. The documentation has been changed in PR 568.

Morpho has decided not to implement any logic that would allow the Morpho Blue owner to disable already enabled `IRM` or `LLTV` that would allow anyone to create broken markets from the beginning.

3.1.4 First borrower of a Market can stop other users from borrowing by inflating `totalBorrowShares`

Severity: High Risk

Context: Morpho.sol#L221-L253

Description: Morpho tracks users' borrowing using shares and share prices similar to ERC4626. SharesMathLib.sol. When a user wants to borrow a certain amount of tokens (`asset`), Morpho contracts mint shares for the user using the formula `assets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares)`. Morpho.sol#L237-L239

Because the calculation rounds down borrowing shares, users can receive shares without incurring actual debt when `totalBorrowAssets` is equal to zero. Attackers can stop other users from borrowing by inflating `totalBorrowShares` without borrowing. Once the `totalBorrowShares` is inflated, the value of `assets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares)` would easily exceed `type(uint128).max` with a normal value (e.g., `1 ether`).

Proof of concept: Here it is demonstrated that an attacker inflates `totalBorrowShares` and prevents other users from borrowing.

```
function testBorrowInflationAttack() public {
    uint amountCollateral = 100 ether;
    _supply(amountCollateral);
    oracle.setPrice(1 ether);
    collateralToken.setBalance(BORROWER, amountCollateral);
    vm.startPrank(BORROWER);
    morpho.supplyCollateral(marketParams, amountCollateral, BORROWER, hex"");
    uint totalBorrowShares = 1e6;

    for(uint i = 0; i < 100; i++) {
        (uint256 returnAssets, uint256 returnShares) =
        morpho.borrow(marketParams, 0, totalBorrowShares - 1, BORROWER, RECEIVER);
        totalBorrowShares += returnShares;
    }
    vm.expectRevert("max uint128 exceeded");
    morpho.borrow(marketParams, 1 ether, 0, BORROWER, RECEIVER);
}
```

This attack vector is exposed when a market is newly created or the `totalBorrowAssets` is low. Attacker can either

1. Back-run a create market transaction and DOS market.
2. Repay all loans in the market and inflates the `totalBorrowShares`.

Recommendation: Consider adding a lower bound of borrowing assets for the first borrower.

Morpho We acknowledge the issue, because:

- It affects only markets with less than 1e4 assets borrowed (checked with this proof of concept) which is extremely rare except at market creation.
- If it becomes a problem at market creation one day, there is an easy fix (but bad UX) consisting on borrowing 1e4 assets when creating a market.
- we didn't find any satisfying fix (I explored an other one, but I'm not happy with it).

Cantina: Acknowledged.

3.2 Medium Risk

3.2.1 Morpho's lack of query for IRM when `totalBorrowAssets == 0` is incompatible with the stateful IRM design

Severity: Medium Risk

Context: Morpho.sol#L455-L470

Description: Morpho calculates accrued interest by querying the `IRM` contract with `Market` and `marketParams`, which include the current market state, such as `totalSupplyAssets`, `totalBorrowAssets`, and `lastUpdate`.

Morpho only queries the `IRM` contract when `totalBorrowAssets` is larger than 0. However, this is incompatible with a stateful IRM, where interest fluctuates and occurs continuously over time. The default `IRM`, SpeedJumpIrm.sol, is a stateful IRM.

Consider the following scenario:

1. At time `t`, the utilization rate is 100% and interest rates are 100% APR.
2. At time `t + 1` to `t + 100 days`, totalBorrowAssets equals zero.
3. At time `t + 101 days`, someone starts borrowing.

According to the `SpeedJumpIrm` design, the utilization rate of the `market` is zero for 100 days, and the interests should be approaching `MIN_RATE` based on `SPEED_FACTOR`. However, since `Morpho` does not query `IRM` when `totalBorrowAssets` equals zero, `SpeedJumpIrm` only updates the interest rates at time `t + 101 days`.

Recommendation: `Morpho` should query IRM even when `totalBorrowAssets` is zero:

```
if (market[id].totalBorrowAssets != 0) {
   uint256 borrowRate = IIrm(marketParams.irm).borrowRate(marketParams, market[id]);
   uint256 interest = market[id].totalBorrowAssets.wMulDown(borrowRate.wTaylorCompounded(elapsed));
   market[id].totalBorrowAssets += interest.toUint128();
   market[id].totalSupplyAssets += interest.toUint128();

   uint256 feeShares;
   if (market[id].fee != 0) {
       uint256 feeAmount = interest.wMulDown(market[id].fee);
       // The fee amount is subtracted from the total supply in this calculation to compensate for the fact
       // that total supply is already increased by the full interest (including the fee amount).
       feeShares =
           feeAmount.toSharesDown(market[id].totalSupplyAssets - feeAmount, market[id].totalSupplyShares);
       position[id][feeRecipient].supplyShares += feeShares;
       market[id].totalSupplyShares += feeShares.toUint128();
   }

   emit EventsLib.AccrueInterest(id, borrowRate, interest, feeShares);
} else {
   IIrm(marketParams.irm).borrowRate(marketParams, market[id]); // update IRM with current market states.
}
```

Note that `SpeedJumpIRM` should handle cases when `totalBorrowAssets` == 0. `IRM` under no circumstances should revert.

Morpho Fixed by PR 574

Cantina: Fixed.

3.3 Low Risk

3.3.1 `asset<>shares` conversion rounding results and possible side effects during repay and withdraw

Severity: Low Risk

Context: Morpho.sol

Description: Depending on the operation that must be performed and depending on the input (asset amount or share amount) Morpho will perform an `asset <> share` conversion that will usually result in favor of the protocol (if you need to repay X number of assets, that amount will be converted in shares rounding down, if you need to repay Y number of shares, it will be converted to assets rounding up, and so on).

This conversion operations, with the rounding, can produce unwanted side effects.

Side effect when a borrower repays by specifying `assets` amount

Let's build this scenario:

1) Alice supply `1000 loanToken`.
2) Bob supply `Y collateralToken` to borrow `100 loanToken` (as much as possible given an LLTV).
3) 10 days pass and interest is accrued.
4) let's assume that in the meanwhile, no one liquidates Bob.

At this point, Bob wants to repay the borrow position in full and needs to know how much `loanToken` has accrued during the delta time (the interest on the borrow amount taken).

Bob queries Morpho by executing `uint256 debtToRepayInAssets = MorphoBalancesLib.expectedBorrowBalance(..., Bob)` and tries to repay his debt in full by executing `morpho.repay(..., debtToRepayInAssets, ...)`

The problem is that the `morpho.repay` function will revert because of an underflow error here:

```
position[id][onBehalf].borrowShares -= shares.toUint128();
```

Why is it reverting?

```
function expectedBorrowBalance(
    IMorpho morpho,
    MarketParams memory marketParams,
    address user
) internal view returns (uint256) {
    Id id = marketParams.id();
    uint256 borrowShares = morpho.borrowShares(id, user);
    (, , uint256 totalBorrowAssets, uint256 totalBorrowShares) = expectedMarketBalances(morpho, marketParams);
    return borrowShares.toAssetsUp(totalBorrowAssets, totalBorrowShares);
}
```

`MorphoBalancesLib.expectedBorrowBalance` query Morpho to retrieve the user's `borrowShares` balance (that is an exact amount). Then it gets the borrow assets/shares balances with the accrued interests and converts the `borrowShares` to assets with `borrowShares.toAssetsUp`

When the user executes the `morpho.repay` function, those assets are converted back to shares with `assets.toSharesDown` because balances are stored in shares.

The problem is that when the second conversion happens, it returns a value that is higher compared to what's stored in `position[id][onBehalf].borrowShares` and because of it the whole tx reverts to an underflow error.

Side effect when a supplier withdraws by specifying `assets` amount

On the withdrawal side, we have the opposite problem.

1) Alice supply `1000 loanToken`.
2) Bob supply `Y collateralToken` to borrow `100 loanToken` (as much as possible given an LLTV).
3) 10 days pass and interest is accrued.
4) let's assume that in the meanwhile, no one liquidates Bob.
5) Jack supply `10_000 loanToken` (to allow Alice to withdraw in full, just for the sake of the example).

Alice wants to withdraw everything she has supplied and query Morpho to get such value by executing `uint256 suppliedBalanceAccruedInTokens = morpho.expectedSupplyBalance(..., alice)`.

At this point, Alice executes `morpho.withdraw(..., suppliedBalanceAccruedInTokens, 0, ...)`. Because of the conversion and because of rounding, while she receives the expected number of tokens back (equal to `suppliedBalanceAccruedInTokens`) but she is left with an amount of "dust" `supplyShares > 0`.

If she executes immediately, `morpho.withdraw(..., 0, dustSupplyShares, ...)` she would burn the remaining dust amount of shares, getting back **zero** loan tokens (this would depend on the state of the total balances of the market, but we can assume that we are in a mature one where the conversion of such dust shares is rounded down to zero tokens).

If she waits enough time, because of the accrual of interest (that increases the value of shares) she will be able to withdraw a dust amount of `loanToken`.

Recommendation: For the borrowing underflow scenario, there's nothing that Morpho can do if not "force" the user to repay by using shares if he wants to be sure to repay an amount of debt without reverting. This issue creates both a UX problem (because the user is forced to repay by specifying shares and not an exact amount of assets) but also a side effect that could end up anyway reverting the transaction if not handled correctly.

If the user does not want to approve an unlimited amount of `loanToken` to Morpho (or the Bundler) but just a fixed amount, the repay transaction risks being reverted anyway. If such transaction is not added in the same block in which the `expectedBorrowBalance` is executed or if the share price changes, the approved amount could end up being not enough to repay the fixed amount of shares that will be burned.

For the withdrawal case, Morpho should document these behaviors to the user that at this point could prefer to withdraw specifying the asset amount instead of the share amount to be able to not lose the dust amount of shares that would remain in the balance after the execution of the operation.

Because Morpho Blue will be a protocol that will be used by EOA via the `Morpho Bundle/Metamorpho` or directly by integrators, Morpho should follow anyway

1) Document these problems and warn user/integrators.

2) Be aware of these limitations and handle them on the UX (when generating the Bundler action) when the user needs to repay/liquidate/withdraw.
3) Warn the integrator and provide them example/best practice on how to approach the repay/liquidate/withdraw execution without reverting.
4) Be aware of this problem to avoid reverting when those operations are executed in the `MetaMorpho` project.

Cantina: Morpho has added documentation support in PR 558.

- `IMorpho.withdraw` and `IMorpho.repay` advise using the `shares` input to fully withdraw or repay the position.
- `MorphoBalancesLib.expectedSupplyBalance` and `MorphoBalancesLib.expectedBorrowBalance` warns the user about the revert case.

3.3.2 Users that borrow as much as possible (up to the LLTV threshold) will be liquidable in the very next block

Severity: Low Risk

Context: Morpho.sol

Description: Unlike protocols like Aave where they have both `LTV` (loan to value) and `LT` (liquidation threshold), the current implementation of Morpho offers just the `LLTV` parameter (Liquidation Loan-To-Value) without any buffer between the amount of debt that the user can take (given a collateral) and the liquidation threshold.

This means that if a user borrows as much as it can borrow (given a collateral and LLTV), such user will be fully liquidable in the very next block as soon as the interest accrual can be triggered.

Recommendation: Morpho should carefully document this behavior and warn the users who perform such operation to be fully aware of the incumbent liquidation event they will incur on the very next block.

A more complex (given the changes needed) approach would be to follow the same path followed by protocols like Aave that introduce a buffer between the amount that the user can borrow (given a collateral) and when the user can be liquidated.

Morpho: Some guardrails will make it difficult for a user to conduct such actions but nothing will be implemented onchain.

3.3.3 `setAuthorizationWithSig` is not fully compliant with the EIP-2612 standard

Severity: Low Risk

Context: Morpho.sol#L423

Description: The EIP-2612 standard defines the `deadline` requirement in the following way

> The current blocktime is less than or equal to `deadline`.

The current implementation of `setAuthorizationWithSig` only allows the execution of the function if the `block.timestamp` is lower than the `deadline` and will revert if those values are equal (not following the standard definition).

Recommendation: Morpho should change the check to follow the specification of the standard and allow executing of `setAuthorizationWithSig` even when `block.timestamp == authorization.deadline`.

Cantina: The recommendations have been implemented in PR 551.

3.3.4 Interest fees will be still accrued even when `feeRecipient` is set to `address(0)`

Severity: Low Risk

Context: Morpho.sol#L129-L135, Morpho.sol#L462-L470

Description: The `feeRecipient` state variable represents the address that will receive all the interest fees accrued across all the markets when the `_accrueInterest` function is executed, interest is accrued and the market's fee attribute is greater than zero.

The current implementation of Morpho Blue allows the owner of the protocol to set the `feeRecipient` to `address(0)`. Even when the such variable is updated to the `address(0)` value, fees across the markets will be accrued and removed from the interest earned by the suppliers.

While Morpho is documenting inside `IMorpho` that the fee recipient can be set to the zero address, it is not explaining why the fees are still accrued even when no one will be able to claim them.

Recommendation: Morpho should consider implementing one of the following actions:

- Prevent the `feeRecipient` to being set to `address(0)`.
- If `feeRecipient` is equal to `address(0)` do not harvest the fees from the interest earned by the suppliers of the markets.
- Document in a clear way why the fees are still accrued and harvest from the interest earned by the suppliers even if those fees won't be claimable by anyone.

Cantina: PR 554 better documents the current `Morpho` contract logic when `feeRecipient` is equal to `address(0)`.

Morpho has decided not to change the implemented logic. When `feeRecipient == address(0)` the fee on the accrued interest will be anyway minted and won't be able to be claimed by anyone.

3.4 Gas Optimization

3.4.1 Gas optimizations

Severity: Gas Optimization

Context: Morpho.sol#L484, Morpho.sol#L500-L502, Morpho.sol#L245-L246

Description/Recommendation:

- Morpho.sol#L484, Morpho.sol#L500-L502: consider passing directly the value of `position[id][borrower].borrowShares` to the `_isHealthy` function to avoid performing 2 SLOADs.
- Morpho.sol#L245-L246: consider swapping the execution of the requirements. Checking `market[id].totalBorrowAssets <= market[id].totalSupplyAssets` cost less gas than performing the `_isHealthy` check.

Morpho: After discussion we'll acknowledge this issue.

3.5 Informational

3.5.1 Consider renaming the `MorphoLib` function in a way that makes it explicit that they are getter functions

Severity: Informational

Context: MorphoLib.sol

Description: At first sight, functions like `supplyShares` and `borrowShares` could be confused with functions that will change the contract's state.

In this case, instead, all the functions offered by the `MorphoLib` library are all getters.

Recommendation: Morpho should consider renaming all the functions in a way that makes it explicit that those functions are getters that do not modify the state.

Here are a couple of suggestions:

- `supplyShares` → `getSupplyShares`
- `borrowShares` → `getBorrowShares`
- `collateral` → `getCollateral`
- and so on...

Morpho: We acknowledge this issue.

3.5.2 Document desired `_accrueInterest` and `IRM` properties

Severity: Informational

Context: Morpho.sol#L456-L459

Description/Recommendation:

parameter	description
A_B	`totalBorrowAssets`
A_S	`totalSupplyAssets`
I	`interest`
r_i	borrow rates buried from `IRM`

`_accrueInterest` has the following property:

$$\Delta t = 0 \vee A_B = 0 \Rightarrow I = 0$$

Now one should document and ask how accrual of interest should perform when split into chunks versus if performed in only one transaction. Let $t_0 < t_1 < t_2$ and in below the exponential function is used although in the implementation of `Morpho` only an approximate function is used that does not have the additive property:

$$A_B \longrightarrow A_B e^{r_2(t_2 - t_0)}$$

$$A_B \longrightarrow A_B e^{r_0(t_1 - t_0)} \longrightarrow A_B e^{r_0(t_1 - t_0) + r_1(t_2 - t_1)}$$

How should $r_2(t_2 - t_0)$ compare to $r_0(t_1 - t_0) + r_1(t_2 - t_1)$? Basically if we break an interest update into multiple chunks how the resulting interests should compare? The answer to this question needs to be document in general or when one is designing and `IRM` system.

- `IrmMock` favours splitting

  Note: fees does not change the result because they just increase the `feeRecipient` share amount

  1) Supplier supply 1000 loan tokens.

2) Borrower supply collateral of 100 collateral tokens.
3) Borrower borrow the max borrowable amount of loan tokens (80 tokens).
4) save the current value of `totalBorrowAssets` for the market.

At this point, I perform a snapshot and branch it:

- Branch 1) warp 365 days, trigger the `_accrueInterest` and see how much has been accrued into the `totalBorrowAssets`. Because there has been no new borrows, it should represent the accrual of interest during the time.
- Branch 2) Perform a loop of 365 iterations where you warp 1 day in the future each time and trigger the `_accrueInterest`. At the end of the loop, check the final value of `totalBorrowAssets`.

It seems to be that by performing the accrual each day, the final accrued amount is greater than performing 1 year warp accrual. The difference is not a rounding error difference, it's about `~0,26752591777469887 ETH`

- **LOG**

```
totalBorrowAssetsStart 80000000000000000000
totalBorrowAssetsEndOneYearSingleJump 86662826664750529120
delta IR 1 year, single jump 6662826664750529120
day passed 365
totalBorrowAssetsStart 80000000000000000000
totalBorrowAssetsEndOneYearMultipleJump 86930352582525227990
delta IR 1 year, multiple jump 6930352582525227990
day passed 365
totalBorrowAssetsEndOneYearMultipleJump > totalBorrowAssetsEndOneYearSingleJump
Delta multiple jump - single jump 267525917774698870
```

- **Test**

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

import "../BaseTest.sol";

contract SLiquidateTest is BaseTest {
    using MathLib for uint256;
    using MorphoLib for IMorpho;
    using SharesMathLib for uint256;

    function testInterestNoFee() public {
        // vm.prank(OWNER);
        // morpho.setFee(marketParams, 0);

        console.log("marketParams.lltv", marketParams.lltv);

        // supplier supply 1000 tokens
        // _supplyAsset(SUPPLIER, 1000 ether);
        _supply(1000 ether);

        uint256 amountBorrowed = 0;
        uint256 amountCollateral = 0;

        // borrower supply collateral
        // let's say I want to borrow 100 ether
        // amountBorrowed = 100 ether;
        // amountCollateral =
↪   amountBorrowed.wDivDown(marketParams.lltv).mulDivDown(ORACLE_PRICE_SCALE, oracle.price());

        amountCollateral = 100 ether;
        amountBorrowed = amountCollateral.mulDivDown(oracle.price(),
↪   ORACLE_PRICE_SCALE).wMulDown(marketParams.lltv);

        vm.startPrank(BORROWER);
        collateralToken.setBalance(BORROWER, amountCollateral);
        morpho.supplyCollateral(marketParams, amountCollateral, BORROWER, hex"");
        morpho.borrow(marketParams, amountBorrowed, 0, BORROWER, BORROWER);
        vm.stopPrank();

        uint256 totalBorrowAssetsStart = morpho.totalBorrowAssets(id);

        uint256 initialTS = block.timestamp;
```

```
            uint256 snapshotId = vm.snapshot();

            // warp 1 year and see how the supply has changed
            vm.warp(block.timestamp + 365 days);

            // trigger the accrual
            vm.startPrank(LIQUIDATOR);
            collateralToken.setBalance(LIQUIDATOR, 1);
            morpho.supplyCollateral(marketParams, 1, LIQUIDATOR, hex"");
            morpho.withdrawCollateral(marketParams, 1, LIQUIDATOR, LIQUIDATOR);
            vm.stopPrank();

            uint256 totalBorrowAssetsEndOneYearSingleJump = morpho.totalBorrowAssets(id);
            console.log("totalBorrowAssetsStart", totalBorrowAssetsStart);
            console.log("totalBorrowAssetsEndOneYearSingleJump",
↪   totalBorrowAssetsEndOneYearSingleJump);
            console.log("delta IR 1 year, single jump", totalBorrowAssetsEndOneYearSingleJump -
↪   totalBorrowAssetsStart);
            console.log("day passed", (block.timestamp - initialTS) / 1 days);

            // revert snapshot
            vm.revertTo(snapshotId);
            // iterate for 365 days
            for (uint256 i = 0; i < 365; i++) {
                vm.warp(block.timestamp + 1 days);
                vm.roll(block.number + 1);

                // trigger the accrual
                vm.startPrank(LIQUIDATOR);
                collateralToken.setBalance(LIQUIDATOR, 1);
                morpho.supplyCollateral(marketParams, 1, LIQUIDATOR, hex"");
                morpho.withdrawCollateral(marketParams, 1, LIQUIDATOR, LIQUIDATOR);
                vm.stopPrank();
            }
            uint256 totalBorrowAssetsEndOneYearMultipleJump = morpho.totalBorrowAssets(id);

            console.log("totalBorrowAssetsStart", totalBorrowAssetsStart);
            console.log("totalBorrowAssetsEndOneYearMultipleJump",
↪   totalBorrowAssetsEndOneYearMultipleJump);
            console.log("delta IR 1 year, multiple jump", totalBorrowAssetsEndOneYearMultipleJump -
↪   totalBorrowAssetsStart);
            console.log("day passed", (block.timestamp - initialTS) / 1 days);

            console.log("totalBorrowAssetsEndOneYearMultipleJump >
↪   totalBorrowAssetsEndOneYearSingleJump");
            console.log(
                "Delta multiple jump - single jump",
                totalBorrowAssetsEndOneYearMultipleJump - totalBorrowAssetsEndOneYearSingleJump
            );
        }
    }
```

- `SpeedJumpIrm` doest not favour splitting

```
uint256 internal constant LN2 = 0.69314718056 ether;
uint256 internal constant TARGET_UTILIZATION = 0.8 ether;
uint256 internal constant SPEED_FACTOR = uint256(0.01 ether) / uint256(10 hours);
uint128 internal constant INITIAL_RATE = uint128(0.01 ether) / uint128(365 days);
```

The test scenario is **identical** to the one posted in the previous point.

In this case, (using `SpeedJumpIrm`) the **final result is inverted**. The final accrued amount of interest is **greater** when the jump is bigger.

```
totalBorrowAssetsStart 8000000000000000000
totalBorrowAssetsEndOneYearMultipleJump 80150077622776977984
totalBorrowAssetsEndOneYearMultipleJump 80150077622776977984

-> Delta single jump / multiple jumps = 653935708421836816
```

The final delta is bigger compared to the previous one and is about `0,653935708421836816 ETH`

– LOG

```
totalBorrowAssetsStart 8000000000000000000
totalBorrowAssetsEndOneYearSingleJump 80804013331198814800
delta IR 1 year, single jump 804013331198814800
day passed 365
totalBorrowAssetsStart 8000000000000000000
totalBorrowAssetsEndOneYearMultipleJump 80150077622776977984
delta IR 1 year, multiple jump 150077622776977984
day passed 365
totalBorrowAssetsEndOneYearMultipleJump < totalBorrowAssetsEndOneYearSingleJump
Delta multiple jump - single jump 653935708421836816
```

- **Test**

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

import "../BaseTest.sol";
import {SpeedJumpIrm} from "src/mocks/SpeedJumpIrm.sol";

contract SLiquidateTest is BaseTest {
    using MathLib for uint256;
    using MorphoLib for IMorpho;
    using SharesMathLib for uint256;

    uint256 internal constant LN2 = 0.69314718056 ether;
    uint256 internal constant TARGET_UTILIZATION = 0.8 ether;
    uint256 internal constant SPEED_FACTOR = uint256(0.01 ether) / uint256(10 hours);
    uint128 internal constant INITIAL_RATE = uint128(0.01 ether) / uint128(365 days);

    function testInterestNoFee() public {
        SpeedJumpIrm sjIRM = new SpeedJumpIrm(address(morpho), LN2, SPEED_FACTOR,
↪  TARGET_UTILIZATION, INITIAL_RATE);
        _setIRM(address(sjIRM));

        console.log("marketParams.lltv", marketParams.lltv);

        // supplier supply 1000 tokens
        // _supplyAsset(SUPPLIER, 1000 ether);
        _supply(1000 ether);

        uint256 amountBorrowed = 0;
        uint256 amountCollateral = 0;

        // borrower supply collateral
        // let's say I want to borrow 100 ether
        // amountBorrowed = 100 ether;
        // amountCollateral =
↪  amountBorrowed.wDivDown(marketParams.lltv).mulDivDown(ORACLE_PRICE_SCALE, oracle.price());

        amountCollateral = 100 ether;
        amountBorrowed = amountCollateral.mulDivDown(oracle.price(),
↪  ORACLE_PRICE_SCALE).wMulDown(marketParams.lltv);

        vm.startPrank(BORROWER);
        collateralToken.setBalance(BORROWER, amountCollateral);
        morpho.supplyCollateral(marketParams, amountCollateral, BORROWER, hex"");
        morpho.borrow(marketParams, amountBorrowed, 0, BORROWER, BORROWER);
        vm.stopPrank();

        uint256 totalBorrowAssetsStart = morpho.totalBorrowAssets(id);

        uint256 initialTS = block.timestamp;
        console.log("initialTS", initialTS);
        uint256 snapshotId = vm.snapshot();

        // warp 1 year and see how the supply has changed
        vm.warp(block.timestamp + 365 days);
        vm.roll(block.number + 1);

        // trigger the accrual
        vm.startPrank(LIQUIDATOR);
        collateralToken.setBalance(LIQUIDATOR, 1);
        morpho.supplyCollateral(marketParams, 1, LIQUIDATOR, hex"");
        morpho.withdrawCollateral(marketParams, 1, LIQUIDATOR, LIQUIDATOR);
        vm.stopPrank();
```

```
            uint256 totalBorrowAssetsEndOneYearSingleJump = morpho.totalBorrowAssets(id);
            console.log("totalBorrowAssetsStart", totalBorrowAssetsStart);
            console.log("totalBorrowAssetsEndOneYearSingleJump",
↪   totalBorrowAssetsEndOneYearSingleJump);
            console.log("delta IR 1 year, single jump", totalBorrowAssetsEndOneYearSingleJump -
↪   totalBorrowAssetsStart);
            console.log("day passed", (block.timestamp - initialTS) / 1 days);

            // revert snapshot
            vm.revertTo(snapshotId);
            // iterate for 365 days
            for (uint256 i = 0; i < 365; i++) {
                vm.warp(block.timestamp + 1 days);
                vm.roll(block.number + 1);

                // trigger the accrual
                vm.startPrank(LIQUIDATOR);
                collateralToken.setBalance(LIQUIDATOR, 1);
                morpho.supplyCollateral(marketParams, 1, LIQUIDATOR, hex"");
                morpho.withdrawCollateral(marketParams, 1, LIQUIDATOR, LIQUIDATOR);
                vm.stopPrank();
            }
            uint256 totalBorrowAssetsEndOneYearMultipleJump = morpho.totalBorrowAssets(id);

            console.log("totalBorrowAssetsStart", totalBorrowAssetsStart);
            console.log("totalBorrowAssetsEndOneYearMultipleJump",
↪   totalBorrowAssetsEndOneYearMultipleJump);
            console.log("delta IR 1 year, multiple jump", totalBorrowAssetsEndOneYearMultipleJump -
↪   totalBorrowAssetsStart);
            console.log("day passed", (block.timestamp - initialTS) / 1 days);

            console.log("totalBorrowAssetsEndOneYearMultipleJump <
↪   totalBorrowAssetsEndOneYearSingleJump");
            console.log(
                "Delta multiple jump - single jump",
                totalBorrowAssetsEndOneYearSingleJump - totalBorrowAssetsEndOneYearMultipleJump
            );
        }
    }
```

Morpho Thanks for the detailed explanation ! For the case of the mocked IRM, this is something we are aware of: it is the responsibility of the IRM to handle the compounding if any, and this simple example IRM does not do it. It explains why we find a greater value when manually compounding the interest.

For the SpeedJumpIrm there is something wrong with the current way of handling the compounding. In particular, the computation explained in this comment is wrong, because the utilization can change, even when there is no interaction with the market. Because of that, in this comment when we compute the integral of `borrowRateAfterJump * exp(speed * t)`, neither the `speed` nor the base rate `borrowRateAfterJump` are constant, as they depend on the utilization. When there are no interaction, the interest accrual naturally increases the utilization, so `err` would increase over time, which means that:

- `speed` would increase over time.
- The base rate would increase because of small jumps (`errDelta > 0` at each interaction).

Both of those factors indicate that the interest accrued would be greater with more interactions.

So then, what explains the decrease in the interest accrued with more interactions that we observe in the tests ? I think it has to do with another inaccuracy in the IRM: the previous borrow rate is at 0, so the early return kicks in and we don't account for the decrease in the interest rate when doing only one interaction. Indeed, when we do two interactions (let's say one day in, and then then 364 days later), we get smaller results than with 365 interactions.

3.5.3 Supply liquidity invariant per market

Severity: Informational

Context: Morpho.sol

Description: Below `market[id]`, `marketParams` in the context are fixed.

parameter	description
A_B	`totalBorrowAssets`
S_B	`totalBorrowShares`
A_S	`totalSupplyAssets`
S_S	`totalSupplyShares`
B_L	amount of loan tokens exchanged and currently allocated to the market in the context. Note that this v

Let's look at the following expression:

$$B_L + A_B - A_S$$

endpoint	$\Delta(B_L + A_B - A_S)$	Notes
`supply`	0	-
`borrow`	0	-
`repay`	0 or 1	this is due the fact that we are using `zeroFloorSub` when updating A_B
`supplyCollateral`	0	-
`withdrawCollateral`	0	-
`liquidate`	0 or 1	this is due the fact that we are using `zeroFloorSub` when updating A_B
`flashLoan`	0	-
`_accrueInterest`	0	-

The `repay` and `liquidate` do not preserve the $B_L + A_B - A_S$ value due to the use of `zeroFloorSub` when updating A_B:

$$B_L \to B_L + a$$

$$A_B \to A_B - (a - \epsilon)$$

where $\epsilon \in$
$0, 1$ and

$$A - \left\lceil \frac{s(A+1)}{S+10^6} \right\rceil \in -1, 0, 1, \cdots, A$$

Note that this discrepancy of when $\epsilon = 1$ only happens when afterwards $A_B = 0$. In these cases the `msg.sender` seems like to send `1` token more than necessary which also breaks the invariant of keeping $B_L + A_B - A_S$ constant and equal to 0:

```
market[id].totalBorrowAssets = UtilsLib.zeroFloorSub(market[id].totalBorrowAssets, assets).toUint128();
...
IERC20(marketParams.loanToken).safeTransferFrom(msg.sender, address(this), assets);
```

Recommendation: It might make sense to ask the `msg.sender` to only send a $- \epsilon$ in these situations so that during the whole lifetime of the market we would have:

$$B_L + A_B - A_S = 0$$

or

$$B_L = A_S - A_B$$

With the current implementation we only know:

$$B_L + A_B - A_S \geq 0$$

3.5.4 `mulDivUp` has a smaller acceptable range compared to other libraries

Severity: Informational

Context: MathLib.sol#L33

Description: The implementation of `mulDivUp` might be cheaper in terms of gas but has a smaller acceptable domain for `(x, y, d)`:

$$\left\lfloor \frac{x \cdot y + (d-1)}{d} \right\rfloor$$

Other libraries like `Solmate` use the following implementation:

```
z := add(gt(mod(mul(x, y), denominator), 0), div(mul(x, y), denominator))
```

$$\left\lfloor \frac{x \cdot y}{d} \right\rfloor + \mathbb{1}_{x \cdot y \mod d \neq 0}$$

Recommendation: If a bigger acceptable range is desired implementation from other libraries can be adopted.

Morpho: Yes it's cheaper and the goal was mostly to have more simple math in the code as well.

3.5.5 There might not be enough incentives for liquidators to realise bad debt

Severity: Informational

Context:

- Morpho.sol#L377-L385

Description: From Morpho's shared internal documents:

> Bad debt detection would lead to bad debt realization, which means that there would be an incentive (in terms of gas) from liquidators to leave some debt.

In most cases there is probably an incentive for the liquidators to leave `1` collateral token (`1 wei` of `ETH` for example) such that `position[id][borrower].collateral = 1` to avoid the storage updates below during the bad debt realisation and thus pay less gas. And thus bad debt would not get realised.

```
uint256 badDebtShares;
if (position[id][borrower].collateral == 0) {
    badDebtShares = position[id][borrower].borrowShares;
    uint256 badDebt = badDebtShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
    market[id].totalSupplyAssets -= badDebt.toUint128();
    market[id].totalBorrowAssets -= badDebt.toUint128();
    market[id].totalBorrowShares -= badDebtShares.toUint128();
    position[id][borrower].borrowShares = 0;
}
```

This also means that suppliers are incentivized to liquidate borrowers in a way that set the `position.collateral = 1 wei` (or to a value that will make the liquidation of the updated position not profitable).

By doing this they:

1) Avoid distributing bad debt (they are suppliers, they don't want to make the supply share price decrease).
2) Pay less gas to perform the liquidation process.
3) Create "dead positions" that won't be liquidated. No one, under some threshold, will liquidate that position because the reward is not worth the tx gas price → that bad debt won't be distributed.

At this point, suppliers have two options:

1) Exit the supply position to avoid debt distribution.
2) Liquidate positions in a way to bring the position to a not profitable liquidation (collateral value < tx gas cost).

In the same internal document the following issue has also been discussed:

> Lender avoiding haircut issue: lenders have an incentive to leave the market before the bad debt is realized, even if it's only to come back just after. They have even more incentives than plain liquidators, because they could also liquidate in between. From a game theory point of view, this means that the biggest lender should win the auction and manage to exit in time while other lenders get a haircut.

Recommendation: Liquidators would need to be incentivised for their bad debt realisation, otherwise they can avoid the extra gas costs. A mechanism needs to be introduced to avoid having positions with bad debts or orphaned accounts.

Morpho

> By doing this they
>
> 1. avoid distributing bad debt (they are suppliers, they don't want to make the supply share price decrease)
> 2. pay less gas to perform the liquidation process
> 3. create "dead positions" that won't be liquidated. No one, under some threshold, will liquidate that position because the reward is not worth the tx gas price → that bad debt won't be distributed

Note that:

1. This is only temporary though, only really profitable if they expect that noone will clean the bad debt (see point 3). Lenders also have the incentive to withdraw + realize the bad debt (+ re-supply), which should be a greater issue and has been discussed already.
2. This is unclear from our testing, because of the gas refund when zeroing the slots `position[id][borrower].collateral` and `position[id][borrower].borrowShares`.
3. This still leaves the possibility for a benevolent actor to clean the bad debt. Note that in the worst case, this issue is saying that bad debt is not realized (which is the case for most money markets).

Because of this, and because implementing an additional mechanism to incentivize bad debt realization further would be complex and create problems on its own (how to parametrize this incentive), I'm not sure this issue should be addressed

Cantina: Acknowledged.

3.5.6 `UtilsLib.zeroFloorSub` is not utilised consistently

Severity: Informational

Context: Morpho.sol#L381-L382, Morpho.sol#L275, Morpho.sol#L372

Description: `UtilsLib.zeroFloorSub` is not utilised consistently. In particular when one is liquidation a portion with bad debt to realise this bad debt the following lines are used:

```
uint256 badDebt = badDebtShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
market[id].totalSupplyAssets -= badDebt.toUint128();
market[id].totalBorrowAssets -= badDebt.toUint128();
```

In some edge cases there could be an underflow by only `1`:

$$A - \left\lceil \frac{s(A+1)}{S+10^6} \right\rceil \in -1, 0, 1, \cdots, A$$

In the above $s, S, A \in \mathbb{Z}_{\geq 0}$ and $s \leq S$.

In the other places where `toAssetsUp(...)` and subtraction is used one is using `UtilsLib.zeroFloorSub(...)` to avoid this underflow by `1` scenario.

parameter	description
A	`market[id].totalBorrowAssets`
S	`market[id].totalBorrowShares`
s	`badDebtShares`

Recommendation: To avoid reverting a liquidating that would realise bad debt with an `1` underflow one could utilise `UtilsLib.zeroFloorSub`:

```
market[id].totalBorrowAssets = UtilsLib.zeroFloorSub(market[id].totalBorrowAssets, badDebt).toUint128()
```

and also `market[id].totalSupplyAssets` would need to be updated with the same delta as `market[id].totalBorrowAssets`.

Cantina: Fixed in: PR 557.

3.5.7 Consider refactoring both `expectedSupplyBalance` and `expectedBorrowBalance` from `MorphoBalancesLib` to offer a better DX

Severity: Informational

Context: MorphoBalancesLib.sol#L94-L106, MorphoBalancesLib.sol#L108-L119

Description: The concept of "balance" in the `MorphoBalancesLib` seems to refer to the tuple `(amount, shares)`.

To be more clear and provide a better DX, Morpho should consider the following changes to both `expectedSupplyBalance` and `expectedBorrowBalance`:

- Rename them to include the `user` term (be explicit, even if there's the implicit information given that you are passing the user address as an input parameter).
- Return both the user's amount and shares (the information is already there).

Recommendation: Morpho should consider performing the following refactoring changes to both `expectedSupplyBalance` and `expectedBorrowBalance`:

- Rename them to include the `user` term (be explicit, even if there's the implicit information given that you are passing the user address as an input parameter)
- Return both the user's amount and shares (the information is already there)

A possible pseudocode implementation of those changes could be as following:

```
- function expectedSupplyBalance(IMorpho morpho, MarketParams memory marketParams, address user)
+ function expectedUserSupplyBalance(IMorpho morpho, MarketParams memory marketParams, address user)
      internal
      view
-     returns (uint256)
+     returns (uint256, uint256)
  {
      Id id = marketParams.id();
      uint256 supplyShares = morpho.supplyShares(id, user);
      (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = expectedMarketBalances(morpho, marketParams);

-     return supplyShares.toAssetsDown(totalSupplyAssets, totalSupplyShares);
+     return (supplyShares.toAssetsDown(totalSupplyAssets, totalSupplyShares), supplyShares);
  }

- function expectedBorrowBalance(IMorpho morpho, MarketParams memory marketParams, address user)
+ function expectedUserBorrowBalance(IMorpho morpho, MarketParams memory marketParams, address user)
      internal
      view
-     returns (uint256)
+     returns (uint256, uint256)
  {
      Id id = marketParams.id();
      uint256 borrowShares = morpho.borrowShares(id, user);
      (,, uint256 totalBorrowAssets, uint256 totalBorrowShares) = expectedMarketBalances(morpho, marketParams);

-     return borrowShares.toAssetsUp(totalBorrowAssets, totalBorrowShares);
+     return (borrowShares.toAssetsUp(totalBorrowAssets, totalBorrowShares), borrowShares);
  }
```

Cantina: PR 550 has not implemented the recommended changes, but has better clarified the scope of the functions. The issue can be marked as fixed.

3.5.8 Consider renaming `MorphoBalancesLib.IMorphoMarketStruct` to a more meaningful name

Severity: Informational

Context: MorphoBalancesLib.sol#L13-L15

Description: The `IMorphoMarketStruct` interface implemented inside `MorphoBalancesLib` in reality is just a stripped version of the `IMorpho` interface.

Morpho should consider renaming the `IMorphoMarketStruct` interface to a more meaningful and clear name like `IMorphoLite`.

Recommendation: Morpho should consider renaming the `IMorphoMarketStruct` interface to a more meaningful and clear name like `IMorphoLite`.

Cantina: PR 567 does not implement the recommendations suggested but addresses the issue correctly.

3.5.9 Consider adding a `expectedTotalBorrowShares` for a better DX

Severity: Informational

Context: MorphoBalancesLib.sol

Description: The `MorphoBalancesLib` library, exposes getter functions useful for dApps and integrators. The main purposes of those functions is to be able to query a market with the expected value after interest accrual.

While the borrowing shares amount of a market is not influenced by the accrual of the interest, it would make sense to expose such a function (called `expectedTotalBorrowShares`) to offer a better DX to dApps and integrators that would be able to fetch all the needed data from the same library contract.

Recommendation: Morpho should consider implementing in `MorphoBalancesLib` the function `expectedTotalBorrowShares` to offer a better DX for dApps and integrators.

Morpho: We disagree to create a function which redirects to `MorphoLib.totalBorrowShares(id)`. Not providing it can actually teach developers that total borrow shares are not expected to change between interactions.

3.5.10 The `authorizer` has no way to manually increase the `nonce` and invalidates all the already signed authorizations that have not yet been used but still valid

Severity: Informational

Context: Morpho.sol#L414-L439

Description: The `authorizer` has no way to manually increase the `nonce` and invalidates all the already signed authorizations that have not yet been used but still valid (deadline still ahead of the block timestamp).

Let's make an example:

1) `Bob` sign an `auth` to allow `Alice` to manage the positions.
2) `Bob` discover that `Alice` is a malicious user and would like to revoke such signature.

Now at this point we have different options, but it depends on the path of actions chosen by `Bob`.

Scenario 1) Bob is not very skilled:

3) `Bob` executes `setAuthorization(alice, false)` thinking that this is enough to prevent `alice` to being a manager.
4) `alice` has still the valid signature and executes `setAuthorizationWithSig(...)` becoming a manager and performing an attack.

Scenario 2) Bob is skilled:

3) The only option that `Bob` has to invalidate such a signature is to generate another signature with the same nonce used for the Alice's signature and execute it to increase the `nonces` and so invalidate the one given to `alice`. By doing this, he will "burn" the Alice's signature nonce. The problem with this solution is that Alice could anyway front run Bob's transaction and execute their signature (that would enable her as a manger).

Recommendation: Morpho should consider implementing a utility function that allows the `authorizer` to manually increase the current `nonce` or set the value of the `nonce` to an arbitrary value (greater than the current `nonce` value).

Morpho: We acknowledge the issue for the following reasons:

1. A well-prepared attacker could frontrun the tx.
2. It adds a non-trivial amount of logic in the protocol.
3. Signatures should be signed with short deadlines to avoid the problem.

3.5.11 Morpho Blue is not compliant with the `ERC-3156: Flash Loans` standard

Severity: Informational

Context: Morpho.sol#L401-L410

Description: The current implementation of `Morpho` is not compliant with the ERC-3156: Flash Loans standard.

Recommendation: Consider implementing all the functions needed to be compliant with the ERC-3156 standard, or explicitly document that the protocol won't be compliant with it and for which reason.

Cantina: Morpho has decided to acknowledge the issue and not refactor the flashloan logic to be compliant with `RC-3156`. Morpho has documented such decision in PR 553.

3.5.12 Consider rephrasing in a more clear way how `setFeeRecipient` for the old fee recipient

Severity: Informational

Context: IMorpho.sol#L126-L130, Morpho.sol#L128-L135

Description: When the `setFeeRecipient` is executed, all the non-accrued fees derived by the accrual of interest will be accounted to the `newFeeRecipient`.

In `IMorpho.setFeeRecipient` natspec documentation, such behavior is explained in the following way:

```
/// @dev Warning: The fee to be accrued on each market won't belong to the old fee
recipient after calling this function.
```

Recommendation: Morpho should consider being more explicit about the behavior and add a warning to the old fee recipient that the available fees, not yet accrued, will be accounted to the new fee recipient if he does not take care to manually trigger in some way the `_accrueInterest` function.

Because there is no way to directly trigger the `_accrueInterest` function (which is declared as `internal`) without performing an action that interacts with the market (supplying, repaying, borrowing or liquidating), Morpho should consider implementing an external function that will just trigger `_accrueInterest` for the specified market. Such utility function could be restricted to be executed by only the `feeRecipient` and accept an array of `marketParams`. Each market, on which the interest will be accrued, must be a created and active market.

Cantina: The behavior has been clarified by PR 554.

However, the current implementation of `Morpho` does not provide any public method that allows the current `feeRecipient` to manually trigger the interest accrual (and fee accrual). The `feeRecipient` could trigger a mix of `supplyCollateral(market, 1 wei, feeRecipient, hex"") + withdrawCollateral(market, 1 wei, feeRecipient, feeRecipient)` to trigger the accrual and get the supplied funds back, but this solution is a "workaround" and would cost more gas compared to an ad hoc function that would just trigger the interest and fee accrual.

Morpho should provide a function, executable only by the `feeRecipient`, that trigger `_accrueInterest` for an existing market.

Morpho: We've decided that we'll expose the function publicly. We'll do a separate PR for this.

3.5.13 Consider appending the `msg.sender` (current owner) to all the events emitted inside functions that use the `onlyOwner` modifier

Severity: Informational

Context: Morpho.sol#L90, Morpho.sol#L99, Morpho.sol#L109, Morpho.sol#L125, Morpho.sol#L134

Description: The `owner` of the `Morpho` contract can be arbitrary changed during the lifetime of the `Morpho` contract and could be different compared to the one specified during the deployment of the contract at `constructor` time.

Morpho should consider appending the `owner` information (in this case, it would be equal to `msg.sender`) to all the events emitted inside those function that are auth gated via the `onlyOwner` function modifier.

Note that by using `msg.sender` instead of `owner` is recommended to avoid to tracking the wrong owner during the emission of such event in the `setOwner` function.

Recommendation: Morpho should consider appending the `owner` information (in this case, it would be equal to `msg.sender`) to all the events emitted inside those function that are auth gated via the `onlyOwner` function modifier.

Morpho: We acknowledge the issue since we can still recompute everything offchain and the likelihood of the fee recipient being a casual users of Morpho Blue is near 0.

3.5.14 Consider adding `name` and `version` to the `EIP712Domain` definition

Severity: Informational

Context: ConstantsLib.sol#L17

Description: The `EIP-712` allows to "describe" the domain with the following attributes:

- `string name` the user readable name of signing domain, i.e. the name of the DApp or the protocol.
- `string version` the current major version of the signing domain. Signatures from different versions are not compatible.
- `uint256 chainId` the EIP-155 chain id. The user-agent should refuse signing if it does not match the currently active chain.
- `address verifyingContract` the address of the contract that will verify the signature. The user-agent may do contract specific phishing prevention.
- `bytes32 salt` an disambiguating salt for the protocol. This can be used as a domain separator of last resort.

Morpho has currently defined the `ConstantsLib.DOMAIN_TYPEHASH` with only the `uint256 chainId` and `address verifyingContract` information.

While it's true that protocol designers do not need to include all the attributes listed above, adding to the current `DOMAIN_TYPEHASH` definition the `name` and `version` attribute could help to improve the UX of the interaction with the protocol via a wallet.

The "EIP712 is here: What to expect and how to use it" article from the Metamask team explains how those attributes are extracted and shown to the user to provide a better UX when the user needs to sign such messages.

Recommendation: Morpho should consider adding the attributes `name` and `version` to their `DOMAIN_TYPEHASH` definition to provide a better UX when the user needs to sign messages toward Morpho Blue.

Morpho: We acknowledge this issue for the following reasons:

1. `name` and `version` are not mandatory to comply to EIP712.
2. There will be no upgrade of Morpho Blue, rendering `version` not useful.
3. There are some downsides to provide a `name` since a scammer could provide the same name to scamp users. Removing the name helps focusing users on the address of the contract which is unique.

3.5.15 Missing events for off-chain analysis

Severity: Informational

Context: Morpho.sol#L70

Description:

- Morpho.sol#L70, emitting `EventsLib.SetOwner(newOwner)` is missing.

Recommendation: Introduce new events for emission or use the already defined ones in the context for this issue.

Cantina: The recommendations have been implemented in PR 549.

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

Severity: Informational

Context: IIrm.sol#L14, IIrm.sol#L19, IMorpho.sol, IMorpho.sol#L55-L56, IMorpho.sol#L75-L86, IMorpho.sol#L126-L130, IMorpho.sol#L149-L152, IOracle.sol#L9-L13, IMorpho.sol#L235-L244, IMorpho.sol#L246-L254, IMorpho.sol#L256-L275, IMorphoCallbacks.sol#L37-L41, EventsLib.sol#L85-L90, EventsLib.sol#L92-L100, EventsLib.sol#L102-L118, IOracle.sol#L13, MarketParamsLib.sol#L15-L16, MathLib.sol, SafeTransferLib.sol, SharesMathLib.sol, SharesMathLib.sol#L10-L11, UtilsLib.sol, MorphoBalancesLib.sol, MorphoLib.sol, MorphoStorageLib.sol, Morpho IRM MathLib.sol, Morpho IRM UtilsLib.sol, Morpho Bundlers IMulticall.sol, Morpho Bundlers IMorphoBundler.sol, Morpho Bundlers EthereumPermitBundler.sol#L15-L18, Morpho.sol#L197-L199, Morpho.sol#L232-L234, Morpho.sol#L316-L318

Description: Across the three projects (Morpho Blue, Morpho IRM, Morpho Bundlers) we have found different natspec documentation issues that include missing parameters, typos or in general suggestion to better improve them:

- IIrm.sol#L14: Missing the `@return` parameter and should document that the returned value has `18` decimal precision.
- IIrm.sol#L19: Missing the `@return` parameter and should document that the returned value has `18` decimal precision.
- IMorpho.sol: Add natspec documentation for each `struct`. Multiple functions are missing the natspec documentation for both `@param` and `@return` values.
- IMorpho.sol#L55-L56: Rephrase/better explain the `@dev` warning about "EIP-712 domain separator" and hard forks.
- IMorpho.sol#L75-L86: Document that the returned value (depending on the state of the market and the accrual state) could be "outdated" and not include the accrued interest and fees.
- IMorpho.sol#L126-L130: Morpho should properly document both:
  - When the fee recipient is updated, the "old" fee recipient won't receive the accrued fees that have not yet been accrued.
  - When the fee recipient is updated to `address(0)` and the `market.fee > 0` the fees will still be accrued for the market (reducing the interest earned by the suppliers) but won't be claimable by anyone.
- IMorpho.sol#L149-L152: Should document that the Morpho Blue is always trusting the value returned by the oracle, even if such value is **stale** or **manipulated/spiked**. If the `oracle.price()` does not revert, Morpho will take for granted that the price is valid and trusted without any check on such value.
- IMorpho.sol#L235-L244 + IMorpho.sol#L246-L254 + IMorpho.sol#L256-L275 + IMorphoCallbacks.sol#L37-L41 + EventsLib.sol#L85-L90 + EventsLib.sol#L92-L100 + EventsLib.sol#L102-L118: Consider renaming `assets`, `seizedAssets` and `repaidShares` (in both the interface, functions and event declarations) into `collateral`, `seizedCollateral` and `repaidCollateralShares` or anyway something that clearly specify that such amount is not the `loanToken` but the `collateralToken`.
- IOracle.sol#L13: Missing the natspec `@return` statement.
- MarketParamsLib.sol#L15-L16: The function is missing both the `@param` and `@return` natspec statement.
- MathLib.sol: All the functions are missing the `@notice`, `@param` and `@return` natspec statement.
- SafeTransferLib.sol: All the functions are missing the `@notice`, `@param` and `@return` natspec statement.
- SharesMathLib.sol: All the functions are missing the `@notice`, `@param` and `@return` natspec statement.
- UtilsLib.sol: All the functions are missing the `@notice`, `@param` and `@return` natspec statement.
- MorphoBalancesLib.sol: Some functions are missing the `@notice`, `@param` and `@return` natspec statement.

- MorphoLib.sol: Some functions are missing the `@notice`, `@param` and `@return` natspec statement.
- MorphoLib.sol: Morpho should document that all the `supply` and `borrow` getters could return an "outdated" values that do not include the accrued interest and fees.
- MorphoStorageLib.sol: Some functions are missing the `@notice`, `@param` and `@return` natspec statement. All the `internal constant` variables are missing the proper `@dev` natspec comment (or at least a "normal" comment).
- Morpho IRM MathLib.sol: Some functions are missing the `@notice`, `@param` and `@return` natspec statement.
- Morpho IRM UtilsLib.sol: Some functions are missing the `@notice`, `@param` and `@return` natspec statement.
- Morpho Bundlers IMulticall.sol: Missing both the natspec documentation for the `interface` and the `function` declarations.
- Morpho Bundlers IMorphoBundler.sol: Missing the natspec documentation for the `interface` declaration.
- Morpho Bundlers EthereumPermitBundler.sol#L15-L18: Missing the natspec documentation for all the input parameters.
- Morpho Bundlers EthereumPermitBundler.sol#L15-L18: Should better document the `allowed` parameter. When the value is equal to `true` the `msg.sender` is giving infinite allowance to the Bundler.
- Morpho.sol#L197-L199 + Morpho.sol#L232-L234 + Morpho.sol#L316-L318: Move the `"No need to verify that onBehalf != address(0) thanks to the authorization check"` comment to the next line given that it is referring to the `require(_isSenderAuthorized(onBehalf), ErrorsLib.UNAUTHORIZED);` instruction.

In general, each `struct` and `enum` defined across the project should be supported by the proper NatSpec documentation that has been introduced with Solidity 0.8.20.

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

Cantina: Morpho has acknowledged the issue.

3.5.17 Document the requirements that the integrators should follow to safely build and deploy Morpho periphery/utils contracts

Severity: Informational

Context: Multiple files across Morpho Blue, Morpho IRM, Morpho Bundlers

Description: Across all the three codebases (Morpho Blue, Morpho IRM, Morpho Bundlers), Morpho is using different Solidity versions and EVM version to build and compile the contracts. The same behavior is used to declare the `pragma` version of some contracts.

Morpho should document in the `README` file of each project which Solidity version and which EVM version should be used to compile and deploy the contracts.

This information is particularly important if contracts will be deployed on chains that currently do not support the same set of features supported by mainnet.

A known issue is the `PUSH0` opcode that has been introduced with `Solidity 0.8.20` and is not currently supported by some of the major Layer 2 chains. Deploying contracts that have been built with such version could introduce unexpected behaviors.

Here are some important examples that should be documented and explained properly:

- Morpho Bundlers is compiled with Solidity 0.8.21, but will use the `evmVersion` "PARIS" (see `hardhat.config.ts`).
- Morpho Blue uses "user-defined value types" that requires at least the `0.8.8` version of Solidity (see `IMorpho.sol`).
- Morpho Blue uses the "`memory-safe`" statement that requires at least the `0.8.13` version of Solidity (see `MarketParamsLib.sol`).

- In general, there are different libraries and interfaces that are declared with a floating `pragma` that support solidity version down to `0.5.x`.

Recommendation: Morpho should extensively document which are the requirements (Solidity version, EVM version) and best practice to build, compile and deploy periphery/utils contracts that will interact with the Morpho ecosystem.