OpenZeppelin | security

# Morpho Labs
# Blue Audit


**September 27, 2023**

# Table of Contents

# Summary

| | |
|---|---|
| Type | DeFi |
| Timeline | From 2023-08-28<br>To 2023-09-11 |
| Languages | Solidity |

| | |
|---|---|
| Total Issues | 19 (3 resolved, 4 partially resolved) |
| Critical Severity Issues | 0 (0 resolved) |
| High Severity Issues | 1 (0 resolved) |
| Medium Severity Issues | 1 (0 resolved) |
| Low Severity Issues | 9 (0 resolved, 4 partially resolved) |
| Notes & Additional Information | 8 (3 resolved) |
| Client Reported Issues | 0 (0 resolved) |

# Scope

We audited the morpho-labs/morpho-blue repository at commit bdcf70a.

In scope were the following contracts:

```
contracts
--
│   Morpho.sol
├── interfaces
│   ├── IERC20.sol
│   ├── IIrm.sol
│   ├── IMorpho.sol
│   ├── IMorphoCallbacks.sol
│   └── IOracle.sol
├── libraries
│   ├── ConstantsLib.sol
│   ├── ErrorsLib.sol
│   ├── MarketsParamsLib.sol
│   ├── MathLib.sol
│   ├── SafeTransferLib.sol
│   ├── SharesMathLib.sol
│   ├── EventsLib.sol
│   └── UtilsLib.sol
│   ├── periphery
│       ├── MorphoBalancesLib.sol
│       ├── MorphoLib.sol
│       └── MorphoStorageLib.sol
```

Additionally, the `Irm.sol` contract has been audited in the morpho-labs/morpho-blue-irm repository at commit bdf8ee0.

# System Overview

The `Morpho` contract is designed as a primitive monolith lending protocol that features complete market isolation and permissionless market creation. It is integrated with user-defined oracles for asset pricing, and interest rate models to manage borrow rates. It also features free flashloans and power delegation by leveraging EIP-712 off-chain signatures.

Market creators can create a market out of any pair of tokens: one will be used as collateral and the other as the borrowable token. They can also choose arbitrary oracle implementations as long as they conform to a specific interface, but the choice of interest rate model is limited to the ones whitelisted by the owner. In the same fashion, the liquidation loan-to-value ratio (LLTV) choice also needs to be enabled by the owner. Once a market is created, everyone is allowed to either:

- Provide and remove borrowable assets liquidity through the `supply` and `withdraw` functions.
- Deposit and withdraw collateral to enable borrowing through the `supplyCollateral` and `withdrawCollateral` functions.
- Borrow and repay arbitrary amounts of `borrowableToken` through the `borrow` and `repay` functions.

Borrowers can borrow up to a maximum defined by the `lltv` parameter and will have to repay an additional interest that is accrued every time any of the mentioned functions (except `supplyCollateral` ) are triggered.

The interest is defined by the `Irm` contract, and the template we audited comes with an error feedback mechanism that continuously tries to bring the utilization rate closer to a given target value. An error variable is used to adjust the borrowing rate at every interaction. The math behind it is complex, but it is worth mentioning that the adjustment is made in a way that avoids sharp changes, both to avoid external manipulations and to not overshoot the adjustment that is needed.

Interest is socialized among all liquidity providers. Eventually, markets can be configured with a fee that only the owner can turn on, on a per-market basis. If a fee is set, a percentage of the accrued interest is subtracted and allocated to a `feeRecipient` instead, in the form of a position with the fee amount as supplied tokens on the relevant market.

In adversarial market conditions, some loans may become unhealthy as their collateral may not be enough guarantee to cover their debt. This will happen when the loan value grows beyond the configured `lltv`. A liquidation mechanism has been set up so that any third party can liquidate an underwater account by repaying their debt in exchange for their collateral assets, at a discount. This discount is known as the incentive factor, which is a percentage capped at 15%. This factor is directly correlated with the expected volatility of the borrowable asset, represented by the configured `lltv` parameter. The lower the `lltv` configured for the asset, the more volatile it is expected to be and thus, a higher incentive for the liquidation is needed.

In case the liquidation itself is not enough to fully liquidate the position, the remaining user debt is bad debt, something that the user will never repay and no liquidation can repay it either. When this happens, this debt is socialized among all liquidity providers by decreasing the total balance of supplied assets. In essence, the liquidity providers take the loss just like they make a profit when interest is accrued. Since the borrowed balance can never exceed the total assets supplied, it is guaranteed that the liquidity providers can absorb all bad debt at all times.

# Security Model and Trust Assumptions

`Morpho` is meant to be a primitive building block for the lending space, so that more complex layers of logic can be added on top of it. These layers may enrich the provided functionality and simplify the user experience for passive investors. That is why the level of assumptions and generalizations for the main `Morpho` contract is of utmost importance. The code docstrings provide a comprehensive overview of the assumptions that are taken in the codebase.

## Privileged Roles

The only privileged roles within the system are the `owner` and users which have been `authorized` by `onBehalf` actors.

The `owner` can:

- Enable specific interest rate models (but once set, they cannot be disabled).

- Enable specific levels of liquidation loan-to-value ratio (but once set, they cannot be disabled either) as long as its value is `< 1e18`.
- Change and/or set a `feeRecipient` address, which will be the beneficiary of generated fees, if activated.
- Activate fees for any market (at a maximum of 25% to be taken from the accrued interest).
- Transfer and renounce ownership.

Authorized users can act on behalf of authorizers to perform the following sensitive actions: `borrow`, `withdraw` and `withdrawCollateral`. However, there is no access control enabled for beneficial actions such as `supply`, `repay` and `supplyCollateral` on behalf of any other user.

# Summary of Findings

We found the codebase to be very well written, with extensive docstrings and technical documentation. We are glad to see robust security patterns in place, and thoughtful considerations being applied to follow best practices.

***Update:*** *The team delivered some changes in the codebase which are not related to any issue highlighted in the report below. For completeness, this is a brief overview of the changes:*

- *In pull request #470, the team renamed the `borrowableToken` variable to `loanToken`. The associated documentation and constants have been adjusted accordingly.*
- *In pull request #455, the team improved some docstrings.*
- *In pull request #503, the team removed an unused import declaration.*
- *In pull request #25, the team added some input validations against potential edge cases in the constructor. Previously, the `Irm` constructor was assumed to be provided with good values since the owner of the protocol has the power to whitelist interest rate models, so it was assumed that proper controls were made before whitelisting one of the `Irm` instances. Now, the target utilization rate must be strictly larger than zero and strictly lower than `WAD`.*
- *In pull request #21, a new event has been added to log the new borrow rate every time it is updated.*

- *In pull request #17, the visibility of a view function has been changed from `public` to `external` since it was not used within the code itself. Additionally, some docstrings have been improved.*

# High Severity

## H-01 Bad Debt Socialization Can Be Maliciously Skipped

In the `Morpho` contract, the `liquidate` function is meant to allow liquidators to repay borrowers' loans at a discount, given by a liquidation incentive factor, deleting their unhealthy positions and in exchange receiving their collateral.

Liquidators can choose the exact amount of collateral they want to seize, which translates into a specific amount of debt to repay, or the exact amount of debt shares to repay, which also translates to a certain amount of collateral to seize. It is up to them to decide whether to perform a partial or a full liquidation by customizing the `seizedAssets` or the `repaidShares` parameters.

A liquidatable position may sometimes generate bad debt, which is debt that will never be repaid by anyone. When this happens, the loss is socialized among all suppliers equally.

Consider a scenario where there exists a market with the following configuration:

- WETH is the collateral token of choice.
- USDC is the borrowable token.
- LLTV of 80%
- Incentive factor of 6%.
- Alice supplies 5,000 USDC, and she will be the victim.
- Bob supplies 5,000 USDC, and he will be the attacker.
- Charlie deposits 4 ETH as collateral and borrows the maximum amount available of 6400 USDC.
- ETH price is $2,000. Notice that the oracle will return the price of one unit of collateral (1 wei) in terms of borrowable token units (USDC) scaled by `1e36`. So in this case the oracle will return `2000e6 * 1e36 / 1e18 = 2000e24`.

If the price of ETH suddenly drops to 1,500 USDC, Charlie's position becomes unhealthy, leaving bad debt behind even if fully liquidated. A liquidator will notice this unhealthy position and go ahead and liquidate it by seizing the full 4 ETH collateral from Alice and repaying 5240 USDC worth of debt, calculated in `repaidAssets`. The remaining 760 USDC is socialized evenly among existing suppliers.

Consider an alternative scenario where Bob, one of the existing suppliers, realizes there is going to be some bad debt generated by this position and wants to escape its socialization. He decides to go ahead and liquidate Charlie's position himself, by seizing exactly the full position collateral minus 1 wei. When doing so, the final collateral on Charlie's position will be exactly 1 wei, bypassing the condition looking for bad debt to socialize.

At this point, Bob decides to use the liquidation callback to perform the following actions:

- Withdraw his supplied amount. There is enough liquidity to do so and no bad debt has been socialized yet.
- Call liquidate on Charlie's position again, using 1 wei as the value for `seizedAssets`. This time the liquidation fully removes Charlie's collateral and thus, the entire bad debt is socialized among all suppliers. However, Bob is not a supplier anymore, so the entire bad debt is allocated to Alice.
- Now that the bad debt socialization has been successfully applied, Bob decides to resupply the same amount of USDC he had initially, leaving his position unaltered by the bad debt socialization. Alice has suffered the entirety of the loss, so effectively Charlie has managed to steal USDC from Alice by redirecting the loss towards her.

This attack was made possible because the current implementation only socializes debt when a position is fully liquidated, but not on partial liquidations. You can find a PoC test that simulates this here. Notice that in this particular case, the PoC doesn't use liquidation's callback for the sake of simplicity.

Consider revisiting the liquidation mechanism design and deciding whether both liquidations should be allowed. If partial liquidations are allowed, consider the possibility of partial bad debt socialization on every liquidation, provided that the position is so unhealthy that it will generate bad debt in any case at current market prices.

As a side note, if a perfect liquidation took place where both collateral and borrow shares are brought down to exactly 0, this check will be true and thus trigger the bad debt socialization mechanism since it only checks for collateral to be zero. However, borrow shares are not checked to be non-zero. If those borrow shares are zero, `badDebt` will be rounded up to 1 wei and 1 wei of assets will be removed from the `totalBorrowAssets` and `totalSupplyAssets` but no borrow shares will be subtracted, effectively condoning 1 wei of borrowers' debt at the expense of the suppliers.

Consider modifying the condition to socialize debt to ensure that both the collateral is zero and that borrow shares are positive, in order to avoid internal accounting errors versus the actual balances held by the contract.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *Although theoretically possible, in practice, the likelihood of such an event is very low. This is because liquidators and lenders have different roles and behaviors. Liquidators are professionals who use optimized bots to act faster than other liquidators, while lenders are more passive and cannot compete in terms of speed. Additionally, according to Morpho's specification, the realization of bad debt should occur as quickly as possible. The issue raised further increases the likelihood of this happening, as lenders also have an incentive to realize the bad debt. This is beneficial for the protocol, as future lenders can enter the market without the fear of bearing the bad debt of previous lenders. Therefore, we acknowledge this issue.*

# Medium Severity

## M-01 Fees Can Be Avoided by Creating New Markets

`Morpho` markets charge an interest rate to borrowers for their services, which is paid out evenly to all the liquidity providers who are supplying those tokens to be borrowed. `Morpho` has the ability to activate fees on a per-market basis, and those fees are taken out of the accrued interest.

This fee collection mechanism effectively charges the liquidity providers exclusively, since whether this fee is charged is completely transparent to the borrowers. If liquidity providers see that a fee has just been configured for their market of choice, they are incentivized to create a brand new market, where the fee is by default set to zero, and migrate their liquidity to the new market as soon as it becomes available. As liquidity drains out, borrowers will be incentivized to borrow from the new market instead.

Liquidity providers can create the same market over and over, with the exact same parameters. The reason they can do so is that the oracle address is arbitrary, so multiple different proxies can be created, all pointing to the same actual oracle implementation of choice. When doing so, the oracle address will be different (even though the implementation is the same), which will produce a unique hash for the `Id`.

Consider determining whether this is the intended behavior. Particularly, consider deciding whether it makes sense for identical markets to be able to exist with only different addresses

for the oracle. Alternatively, consider restricting the functionality by having oracles being whitelisted, similar to what already happens to interest rate models.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *The fee on a specific market should be activated only if necessary and if the market's size is significant enough so that a very small fee does not affect lenders and borrowers. We acknowledge this issue.*

# Low Severity

## L-01 Multiple Pragma Statements Span Several Minor Versions of Solidity

Using a pragma statement that spans several minor Solidity versions can lead to unpredictable behavior due to differences in features, bug fixes, deprecation, and compatibility between minor versions.

Throughout the codebase there are `pragma` statements that span several minor and/or outdated versions of Solidity. For instance:

- The `pragma` statement on line 2 of `IERC20.sol`
- The `pragma` statement on line 2 of `IIrm.sol`
- The `pragma` statement on line 2 of `IMorpho.sol`
- The `pragma` statement on line 2 of `IMorphoCallbacks.sol`
- The `pragma` statement on line 2 of `IOracle.sol`

Consider pinning the Solidity version more specifically throughout the codebase to ensure predictable behavior and maintain compatibility across various compilers. Moreover consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase.

***Update:*** *Partially resolved. The Solidity versions were unified but lowered to at least 0.5.0 in pull request #502 at commit b2c9260 for interfaces and at least 0.8.0 in* `MarketParamsLib` *and* `SafeTransferLib`*. The reasoning relies on integrators using old Solidity versions that would otherwise have to manually copy their code and/or change their projects' compiler versions.*

## L-02 `require` Statements With Multiple Conditions

Throughout the codebase there are `require` statements that require multiple conditions to be satisfied. For instance:

- The `require` statement on line 428 of `Morpho.sol`
- The `require` statement on line 23 of `SafeTransferLib.sol`
- The `require` statement on line 29 of `SafeTransferLib.sol`

To simplify the codebase and raise the most helpful error messages for failing `require` statements, consider having a single `require` statement per condition with its own error message when applicable.

***Update:*** *Partially resolved in pull request #482 at commit 75d1b48. The Morpho team decided to split the `SafeTransferLib` library's double requirement and keep the signature verification requirements unchanged.*

## L-03 Missing Docstrings

Throughout the codebase there are several parts that do not have docstrings. For instance:

- Lines 8, 9 and 10 in `SafeTransferLib.sol`
- Lines 14 and 15 in `MorphoBalancesLib.sol`
- Lines 7 in `MorphoLib.sol`
- Lines 4 in `ErrorsLib.sol`
- Lines 9 in `MathLib.sol`
- Lines 4 in `UtilsLib.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API but also of libraries even if those are straightforward. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

***Update:*** *Partially resolved in pull request #493 at commit df4db09 and in pull request #31 at commit 9e25203. The `IMorphoMarketStruct` interface in `MorphoBalancesLib` and the `IERC20Internal` interface within the `SafeTransferLib` are left without docstrings.*

## L-04 Precision Loss Due to Decimals Management

The `IOracle` interface states that the price returned by the oracle will be the quote of 1 unit of collateral expressed in borrowable token decimals, scaled by 36 decimals. Consider the following scenario:

- 1 WETH is worth 2,000 USDT
- 1 USDT is worth exactly 1 USD
- WETH has 18 decimals and USDT has 6 decimals

In this scenario, if the borrowable token is WETH, the oracle will return `5e44` but if the borrowable token is USDT, then the amount returned will be `2e27`.

The `price` function of the oracle is used in the `_isHealthy` function to determine the maximum borrowable amount.

In the aforementioned example, and assuming an `lltv` value of 100% (it has to be strictly lower than 100%, but for simplicity let's assume it is exactly 100%), any `collateral` amount lower than `5e8` will be zeroed out by the division by `ORACLE_PRICE_SCALE`. If `lltv` is actually lower than that, an even bigger collateral amount might be truncated down to 0, making it unusable for any loan. It is unclear whether these edge cases might cause token dust to accumulate over time, and become stuck in the contract forever.

The problem might be even worse in more exotic scenarios, like using highly-priced low decimals tokens like WBTC (8 decimals) and high decimals stablecoins such as DAI (18 decimals).

Even though the outcome is different, this truncation also happens within the `liquidate` function, but we could not find any profitable attack vector around it.

In order to avoid these scenarios, consider deciding whether the oracle price scaling factor should be a function on the decimals delta between both tokens rather than a fixed value of 36. Additionally, consider expanding the test suite to account for a diverse set of token decimals for both the borrowable and the collateral tokens, since they are both currently set to 18 decimals by default for all test cases.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *The loss of precision mentioned in the example is expected since 5e8 wei of WETH is worth 0 wei of USDT in practice. Besides that, even though the oracle scale price is fixed to 1e36, the price returned by the oracle must have a precision of `36 +`*

> *`borrowable decimals - collateral decimals` as documented in the IOracle file taking into account the different decimals of tokens.*

## L-05 Replay Attacks Are Possible After Hardforks

If the event of a hard fork of the underlying chain that the `Morpho` contract is deployed to, the `EIP712_DOMAIN_TYPEHASH` value will become exploitable. This will happen because the `chainId` parameter is computed in the constructor. If a hard fork takes place after the contract's deployment, the `chainId` value used to compute the EIP-712 domain typehash will not be modified, so any signature used on the main chain will be replayable on the forked chain.

Consider checking the current `chainId` value every time against a cached value generated on the constructor in order to use the new one in case a hard fork is detected.

***Update:*** *Acknowledged, not resolved. The Morpho team added more explicit docstrings to document the behaviour on pull request #484 at commit 8780d93, but no explicit resolution has been applied.*

## L-06 Missing Assumptions

As stated in the introduction, the codebase already provides a comprehensive list of assumptions made by the code. However, we identified some potential threats that should be added as additional assumptions:

- Tokens with double entry points will enable the creation of different markets with the same tokens. There might be other edge cases but we were not able to detect a specific threat.
- If `owner` and/or `feeRecipient` are set to be a contract, they must conform to the necessary interfaces to deal with `owner`/`feeRecipient` compatible actions.
- Meta transactions are not supported since plain `msg.sender` is used.
- EIP-1271 is not supported, so this version of `Morpho` will not be compatible with account abstraction.
- If `Morpho` gets blacklisted by USDC or USDT, all related markets will break at once since no tokens can be moved by then.

Consider whether necessary actions need to be taken in order to remove any of these assumptions, or alternatively, if those should be clearly stated within the docstrings to at least raise awareness about them.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *We acknowledge this issue. The last point is covered by the comment "The token can revert on transfer and* `transferFrom` *for a reason other than an approval or balance issue." The other points are either edge cases that do not break Morpho or are simply evident by reading the code (EIP-1271 support). As there are numerous cases like that we prefer not to mention them for clarity.*

## L-07 Lack of Input Validation

There are some instances in the codebase where the input is not correctly sanitized:

- In the `Morpho` contract, the `withdraw`, `borrow` and `withdrawCollateral` functions have a `receiver` parameter that is not checked to not be `address(this)`. If that happens, any tokens sent to `Morpho` will be stuck forever, since it has no functionality to handle tokens transferred directly into it, as it never relies on `balanceOf(address(this))`.
- Markets may be created without an oracle by using the zero address, since there is no validation on its address. Even though there are assumptions that it can revert, there is no mention of unset oracles, which will prevent sensitive actions such as liquidations or collateral withdrawals. Consider checking that the specified oracle has at least code deployed on its address, to avoid setting externally owned account addresses instead.
- When setting the fees, the `feeRecipient` is not checked to be set. If it is not set, fees will be assigned to a supply position owned by the zero address and it will be impossible to rescue them.

Consider evaluating each one of these examples and deciding whether they should be fixed with additional checks on the input provided.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *We acknowledge this issue for the following reasons:*
>
> - *Funds can be set to many other addresses where funds could be stuck so adding a comment stating that funds would be stuck if* `address(this)` *is passed as the receiver is not complete.*
> - *These cases are implied by the liveness section where calls should not fail on the IRM, tokens and oracles.*
> - *This is acknowledged in the code and won't add changes to the code.*

## L-08 New Markets Are Susceptible to Griefing Attacks

When new markets are created, there are no assets supplied nor any collateral provided to borrow against them. In order to provide assets to be borrowed, liquidity providers call the `supply` function to earn interest over borrows of the same assets, triggered by borrowers calling the `borrow` function instead. For a borrower to initiate a loan, they must provide enough collateral first by calling the `supplyCollateral` function.

If liquidity providers want to exit their positions, they will call the `withdraw` function, while borrowers can repay their loans by using the `repay` function. The `withdraw` function has a check in place to avoid letting anyone withdraw more assets than the supplied ones. Notice that the limit is imposed by the collateral value and not by a percentage of the total amount of supplied assets. This means that one can borrow 100% of the supplied assets if enough collateral is provided taking into account the loan-to-value ratio.

Unfortunately, liquidity providers can become victims of a griefing attack by a malicious actor who may not want them to remove their liquidity. This can be achieved by sandwiching the `withdraw` call of a liquidity provider by first borrowing enough to cover 100% of the supplied assets minus the percentage withdrawn by the liquidity provider plus one, and then back-running the failed liquidity provider transaction to repay the loan within the same block, avoiding any interest on the short-lived borrow. This way, the check within the `withdraw` function will be triggered and any attempt to remove the liquidity will fail.

Conversely, the `borrow` function has a check that avoids taking out a loan if the total amount of borrowed assets exceeds the amount of supplied ones. In this scenario, malicious liquidity providers might exit their positions and re-join the protocol in a sandwich attack to force the `borrow` call to fail. This is less severe than preventing liquidity providers from exiting their positions as we could not find good incentives for attackers to do so. However, we are reporting it for completeness.

Both issues are unlikely to happen in mature markets with high volumes. However, they are extremely easy to perform in recently created markets. If it happens, liquidity providers will not be able to withdraw their funds until the attack stops.

Consider documenting such behaviour or, alternatively, consider whether a certain minimum amount of supplied assets should be achieved before allowing any borrowing operation. Notice that on Ethereum mainnet, services like Flashbots / MEVBlocker can protect from front-running (and users should be encouraged to use it), however, such services are not yet available in all

chains that feature public mempools and unfortunately not all users are even aware of their existence.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *We acknowledge this issue since this attack is quite unlikely to occur and there is no simple way to prevent it without introducing some timelocks adding more complexity than anything.*

## L-09 Rounding Error Might Prevent Borrowing at the Maximum Limit

According to the code and documentation, a position is considered healthy if the total borrowed amount is equal to its maximum borrowable amount.

When a borrower decides to take such a loan providing exact `assets`, that amount gets rounded up in order to calculate the equivalent amount of shares. After performing all the necessary internal accounting changes to account for the new loan, a safety check is performed to ensure that the resulting position is healthy.

The issue is that when determining if the newly created position is healthy, the position's `borrowShares` value is read from storage and then converted back to assets by rounding it up again, causing a double round-up error which results in the calculated `borrowed` value being 1 wei larger than the actual borrowed amount and thus, incorrectly flagging the position as unhealthy. Notice that this depends on the real numbers being used, which might be affected by current oracle prices and might show a randomized behaviour.

Consider determining whether loans taken at the maximum borrowable amount are considered healthy. If they are, consider avoiding the double round-up. Alternatively, consider removing the equality from the condition that determines whether a position is healthy.

***Update:*** *Partially resolved in pull request #489 at commit 465cb7f. The behavior was left unchanged since the suggested fix only shifts the result by one wei, but the Morpho team added explicit docstrings to document it.*

# Notes & Additional Information

## N-01 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the clarity of the code, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Within `Morpho.sol`, line 16 has a global import. This is because the `ConstantsLib.sol` is inconsistent with the rest of the files in the same directory which are well-encapsulated within a `library` definition.

Following the principle that clearer code is better code, consider defining `ConstantsLib` within a `library` definition and using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *This non-explicit import is only for one import (`ConstantsLib`). Adding the `ConstantsLib` prefix would make the code much more verbose. Thus, we acknowledge this issue.*

## N-02 Variable Cast Is Unnecessary

Within the `Morpho` contract, the `irm` variable in the `enableIrm` function is unnecessarily cast.

To improve the overall clarity, intent, and readability of the codebase, consider removing this unnecessary cast.

***Update:*** *Resolved in pull request #497 at commit 6108eac.*

## N-03 Inconsistent Use of Named Return Variables

In the `IMorpho` interface, some functions return named parameters while others do not.

In order to improve the overall readability and quality of the codebase, consider enforcing a consistent style regarding return variables across all files.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *In most cases, names are added to remove ambiguity and make the order of outputs clear to developers and integrators. When there is a single variable that is returned the benefit is non-existent or it makes the code even more verbose. Thus, we acknowledge this issue.*

## N-04 Unused Import

In `Irm.sol` the import `WAD_INT` is unused and could be removed.

Consider removing unused imports to improve the overall clarity and readability of the codebase.

***Update:*** *Resolved in pull request #30 at commit 93a54e8.*

## N-05 Duplicated Code

In the `Morpho` contract, there are many checks that are repeated in several places and can be isolated into internal functions or modifiers in order to reduce the code size and overall quality. Some examples are:

- The market existence check.
- The check on whether only one between `shares` or `assets` is being provided as input.
- The check on whether the `receiver` is the zero address.

This can lead to issues later on in the development lifecycle, and leaves the project more prone to the introduction of errors. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. Consider determining whether it makes sense to consolidate them into one single place instead of repeating them.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *Given the code is very simple, it has the advantage of being very explicit so we prefer to keep the current state of the code.*

## N-06 Use of Hardcoded Numbers

The `MarketParamsLib` is a library intended to parse information about specific market parameters represented by the `MarketParams` struct.

If the `MarketParams` data structure ever changes, the number `5` in the assembly block will need to be updated too. In order to avoid error-prone patterns, consider defining the length needed in a constant so that if it is ever meant to be used somewhere else or updated, it will be easier to remember it.

***Update:*** *Resolved in pull request #486 at commits 437ba02 and 1bc4cee.*

## N-07 Market Existence Is Not Checked in `MorphoBalancesLib`

In the `MorphoBalancesLib` library, the majority of the functions rely on the internal `expectedMarketBalances`. However, this function calculates the `Market` id but never checks its existence.

Consider whether is worth adding a check to prevent using the library on nonexistent markets.

***Update:*** *Acknowledged, not resolved. The Morpho team stated:*

> *It should be the integrator's responsibility to check whether the market is created.*

## N-08 `SafeTransferLib` Does Not Revert on Zero Address

The `SafeTransferLib` library is in charge of dealing with ERC-20 `transfer` and `transferFrom` calls used within the `Morpho` contract. Given the fact that market creation is a permissionless action, one can create a market where either the `borrowableToken` and/or the `collateralToken` are the zero address. In such case, the library fails in reverting any attempt to call `transfer` or `transferFrom` on the zero address.

This is because the returned data length is checked to be zero but not that the target contract's address is actually a contract with code in it. Note that this will also happen when any of those tokens is set to a non-contract address.

Consider adopting the same pattern used in OpenZeppelin's Address library, where data length is accepted to be zero only if the code size at the target address is not zero.

***Update:*** *Acknowledged, not resolved. The behavior was left unchanged but the Morpho team added explicit docstrings to warn about it in pull request #485 at commit a3d7a87.*

# Conclusion

The audit of the current Morpho Blue protocol version, including the IRM implementation, uncovered a single high-severity, one medium-severity, multiple low-severity, and some noteworthy issues. We found the codebase to be very robust, highly readable, well-modularized and thoroughly documented. The developers made sure to include many assumptions and known behaviours in the docstrings, alongside general usage warnings and justifications of specific operations to facilitate the overall understanding of the protocol. Additionally, the repository contained an exhaustive fuzzing test suite, which highly improves the protocol's security.

While checking for potential dust accumulation within the protocol, we were not able to successfully run the entire test suite since some unit tests were failing on edge fuzzing scenarios. We encourage the team to make sure there is no dust accumulation and that all unit tests run successfully ensuring good levels of coverage. Moreover, it is encouraged to expand the test suite to include markets created with tokens that have a different amount of decimals than the default 18. Those tokens might generate rounding errors and truncation issues that are not clearly visible if all test cases assume default 18-decimal tokens.