# Morpho Blue IRM

## Security Review


Cantina Managed review by:

**Saw-mon-and-Natalie**, Lead Security Researcher

**Jonah1005**, Lead Security Researcher

**StErMi**, Security Researcher



November 15, 2023

# 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** | *Must* fix as soon as possible (if already deployed). |
| **High** | Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users. |
| **Medium** | Global losses <10% or losses to only a subset of users, but still unacceptable. |
| **Low** | Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies. |
| **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. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

## 2 Security Review Summary

IRM

From Sep 28th - Oct 16th the Cantina team conducted a review of Morpho Blue IRM on commit hash 6a6cc5...da8078. The team identified a total of **9** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 2
- Medium Risk: 1
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 5

# 3 Findings

## 3.1 High Risk

### 3.1.1 `avgBorrowRate` can blow-up for small `linearVariation`

**Severity:** High Risk

**Context:**

- SpeedJumpIrm.sol#L149

**Description:** For small values of `linearVariation`, the average borrow rate calculated has a really big error which causes the result to really deviate from the expected value. These errors come from amplified division errors divided by `linearVariation`.

The exponential function has this property that for small x (terms/operations below are not in `WAD`):

$$\frac{e^x - 1}{x} \approx 1 + \frac{x}{2}$$

The `wExp(x)` defined in MathLib has the following property for small x (up to division errors) ( $|x| < \lfloor \frac{}{2} \rfloor$ ):

$$\frac{\texttt{wExp}(x) - 10^{18}}{x} \cdot 10^{18} = 10^{18} + \frac{x}{2}$$

Above is due to the fact that $\texttt{q} = \lfloor \frac{\texttt{x} + \texttt{roundingAdjustment}}{}$

$$V = \texttt{wExp}(x) = 10^{18} + x + \lfloor \frac{\lfloor \frac{x^2}{10^{18}} \rfloor}{2} \rfloor = 10^{18} + x + \frac{x^2}{2 \cdot 10^{18}} - \epsilon_2$$

$$r_{avg} = \lfloor \frac{r_n - A}{x} \cdot 10^{18} \rfloor = \lfloor \frac{\frac{A \cdot V}{10^{18}} - \epsilon_0 - A}{x} \cdot 10^{18} \rfloor$$

$$= \left\lfloor \frac{\frac{A \cdot (10^{18} + x + \frac{x^2}{2 \cdot 10^{18}} - \epsilon_2)}{10^{18}} - \epsilon_0 - A}{x} \cdot 10^{18} \right\rfloor$$

$$= \left\lfloor A(1 + \frac{x}{2 \cdot 10^{18}}) - \frac{A\epsilon_2 + 10^{18}\epsilon_0}{x} \right\rfloor = \left\lfloor \frac{A(10^{18} + \frac{x}{2})}{10^{18}} - \frac{A\epsilon_2 + 10^{18}\epsilon_0}{x} \right\rfloor$$

$$= \left\lfloor \frac{A(10^{18} + \lfloor \frac{x}{2} \rfloor)}{10^{18}} + \frac{A\epsilon_3}{10^{18}} - \frac{A\epsilon_2 + 10^{18}\epsilon_0}{x} \right\rfloor$$

$$= \left\lfloor \frac{A(10^{18} + \lfloor \frac{x}{2} \rfloor)}{10^{18}} \right\rfloor + \left\lfloor \epsilon_4 + \frac{A\epsilon_3}{10^{18}} - \frac{A\epsilon_2 + 10^{18}\epsilon_0}{x} \right\rfloor$$

and so if we let:

$$E = \left\lfloor \epsilon_4 + \frac{A\epsilon_3}{10^{18}} - \frac{A\epsilon_2 + 10^{18}\epsilon_0}{x} \right\rfloor$$

Then:

$$r_{avg} = \texttt{wMulDown}(A, 10^{18} + \lfloor \frac{x}{2} \rfloor) + E$$

Note that without division errors one would have calculated the $r_{avg}$ as $A(1 + \frac{x}{2 \cdot 10^{18}})$. When $x$ or the `linearVariation` is really small the error part E blows up due to the terms $-\frac{A\epsilon_2 + 10^{18}\epsilon_0}{x}$.

So for small values of $x$ , we can define $r_{avg}$ to be:

$$r_{avg} = \texttt{wMulDown}(A, 10^{18} + \lfloor \frac{x}{2} \rfloor)$$

or even better:

$$r_{avg} = \texttt{mulDivDown}(A, 2 \cdot 10^{18} + x, 2 \cdot 10^{18})$$

This second form would reduce the E term further by eliminating the $\frac{A\epsilon_3}{10^{18}}$ component.

PoC:

```
function testBorrowRateCase1() public {
    Market memory market0;
    market0.totalBorrowAssets = 200 ether;
    market0.totalSupplyAssets = 1_000 ether;
    market0.lastUpdate = uint128(block.timestamp);

    uint256 avgBorrowRate = irm.borrowRate(marketParams, market0);
    market0.totalBorrowAssets = 800 ether - 10000000000;
    uint timestamp = block.timestamp;
    market0.lastUpdate = uint128(timestamp);
    timestamp += 12;
    vm.warp(timestamp);
    vm.roll(block.number + 1);

    (uint256 prevBorrowRate,) = irm.marketIrm(marketParams.id());
    console2.log("prevBorrowRate:", prevBorrowRate);
    avgBorrowRate = irm.borrowRateView(marketParams, market0);
    console2.log("avgBorrowRate:", avgBorrowRate);
}
```

```
prevBorrowRate:         317097919

err:                    -12500000
errDelta:               749999999987500000  ~ 0.7xxx ethers

speed:                  -3
linearVariation:        -36
jumpMultiplier:         1683454723075656906 ~ 1.6xxx ethers
variationMultiplier:    999999999999999964  ~ WAD which makes sense and acts as 1

borrowRateAfterJump:    533819989
newBorrowRate:          533819988
avgBorrowRate:          27777777777777777
```

`newBorrowRate - borrowRateAfterJump = -1`

```
avgBorrowRate = (-1 * WAD) / (-36) = 27777777777777777 ~ 0.0277... ethers
```

**Recommendation:** When x is small:

$$|x| < \lfloor \frac{}{2} \rfloor$$

calculate $r_{avg}$ as:

$$r_{avg} = \texttt{mulDivDown}(A, 2 \cdot 10^{18} + x, 2 \cdot 10^{18})$$

Here is a rough solidity implementation:

```
if (
    (-LN2_INT / 2 < linearVariation) &&
    ( LN2_INT / 2 > linearVariation)
) {
    avgBorrowRate = MorphoMathLib.mulDivDown(borrowRateAfterJump, uint256(int256(2 * WAD) + linearVariation),
    ↪  2 * WAD);
}
else {
    avgBorrowRate = uint256((int256(newBorrowRate) - int256(borrowRateAfterJump)).wDivDown(linearVariation));
}
```

Note that `borrowRateAfterJump` should always be non-negative and the cast `uint256(int256(2 * WAD) + linearVariation)` should be safe since:

$$|x| < \lfloor \frac{}{2} \rfloor < 10^{18}$$

**Morpho**

**Cantina:**

### 3.1.2 Incorrect upper bound check in `wExp(x)` can produce an overflowed result

**Severity:** High Risk

**Context:**

- MathLib.sol#L26

**Description:** Upper-bound used in `wExp(x)` is not restrict enough:

```
// Revert if x > ln(2^256-1) ~ 177.
require(x <= 177.445678223345999210 ether, ErrorsLib.WEXP_OVERFLOW);
```

As this function accepts `x` in the `18` decimal format and supposed to return an `18` decimal number the upper bound should be calculated similarly to `remco`'s `FixedPointMathLib`:

$$\lfloor 10^{18} e^{\lfloor \frac{x+\epsilon}{10^{18}} \rfloor} \rfloor \leq 10^{18} e^{\frac{x+\epsilon}{10^{18}}} \leq 2^{256} - 1$$

so:

$$x \leq 10^{18} \ln\left(\frac{2^{256}-1}{10^{18}}\right) - \epsilon$$

here $\epsilon =$

## 3.2 Medium Risk

### 3.2.1 `avgBorrowRate` is miscalculated as zero when `newBorrowRate` equals to `borrowRateAfterJump`

**Severity:** Medium Risk

**Context:** SpeedJumpIrm.sol#L147

**Description:** The `SpeedJumpIrm` calculates the interest rate based on SpeedFactor and JumpFactor. According to the document, at any time `t`, the level of the rate is given by the formula:

$$r(t) = r(\text{last}(t)) * \text{jump(t)} * \text{speed(t)}$$

For Morpho blue to calculate the interests during two interactions of IRM, the `SpeedJumpIrm` calculated the `avgBorrowRate` as `uint256((int256(newBorrowRate) - int256(borrowRateAfterJump)).wDivDown(linearVariation))`. `borrowRateAfterJump` is the interests rate applied by the JumpFactor and the `newBorrowRate` is the end interest rates applied by both SpeedFactor and JumpFactor.

In the SpeedJumpIrm implementation, two variables are calculated as follow: SpeedJumpIrm.sol#L134-L139

```
int256 linearVariation = speed * int256(elapsed);
uint256 variationMultiplier = MathLib.wExp(linearVariation);

// newBorrowRate = prevBorrowRate * jumpMultiplier * variationMultiplier.
uint256 borrowRateAfterJump = marketIrm[id].prevBorrowRate.wMulDown(jumpMultiplier);
uint256 newBorrowRate = borrowRateAfterJump.wMulDown(variationMultiplier);

// ...

uint256 avgBorrowRate;
if (linearVariation == 0) avgBorrowRate = borrowRateAfterJump;
// Safe "unchecked" cast to uint256 because linearVariation < 0 <=> newBorrowRate <= borrowRateAfterJump.
else avgBorrowRate = uint256((int256(newBorrowRate) - int256(borrowRateAfterJump)).wDivDown(linearVariation));
```

Since the `speed` variable depends on the `err`of the current utilization rate and the target utilization rate, it can be really small results in `linearVariation` very small or zero. Thus, in the implementation, it returns `avgBorrowRate` as `borrowRateAfterJump` when `linearVariation` is zero.

However, there's an edge case when `linearVariation` is small but not zero. In this case, `borrowRateAfterJump.wMulDown(variationMultiplier);` equals `borrowRateAfterJump` because of rounding down. Resulting in the wrong `avgBorrowRate` at SpeedJumpIrm.sol#L149 since $\frac{0}{\text{linearVariation}} = 0$

**Recommendation:** This issue is highly related to *`avgBorrowRate` can blow-up for small `linearVariation`*. Both describe the approximation error when `linearVariation` is small. Recommend to use a simpler formula to approximate the average borrow rate when `linearVariation` is small. The recommended mitigation of *`avgBorrowRate` can blow-up for small `linearVariation`* works in two cases.

**Morpho:**

**Cantina:**

## 3.3 Low Risk

### 3.3.1 First `err` of a market anchors the range of `errDelta` and `JumpMultiplier`

**Severity:** Low Risk

**Context:**

- SpeedJumpIrm.sol#L122

**Description:** The `SpeedJumpIrm` calculates the interest rate based on SpeedFactor and JumpFactor. According to the document, at any time `t`, the level of the rate is given by the formula:

$$r(t) = r(\text{last}(t)) * \text{jump(t)} * \text{speed(t)}$$

Also, to avoid the manipulation risks (notion document), the jump function is chosen this way to avoid manipulation, as:

$$\forall e, k_D^{\Delta(e)} * k_D^{\Delta(-e)} = 1$$

In the SpeedJumpIrm implementation, the jump factor is defined by the following formula, where the interest rate depends on the `errDelta`. SpeedJumpIrm.sol#L128

```
jumpMultiplier = MathLib.wExp(errDelta.wMulDown(int256(LN_JUMP_FACTOR)));
```

Since `errDelta = err - marketIrm[id].prevErr` and $-1 < err < 1$ , the first *err* to be recorded would set the range of `JumpMultiplier`.

Assume two scenarios where the market stabilizes at the target utilization rate.

- Case 1: the first `err` to be recorded is `1`, and the market stabilizes at the target utilization rate, and `err` is 0.

  $$\textit{jumpMultiplier} = \textit{JumpFactor}^{-1}$$

  , and

  $$\textit{JumpFactor}^0 > \textit{jumpMultiplier} > \textit{JumpFactor}^{-2}$$

  , assume $\textit{JumpFactor} > 1$

- Case 2: the first `err` to be recorded is `-1`, and the market stabilizes at the target utilization rate, and `err` is 0.

  $$\textit{jumpMultiplier} = \textit{JumpFactor}^1$$

  , and

  $$\textit{JumpFactor}^2 > \textit{jumpMultiplier} > \textit{JumpFactor}^0$$

  , assume $\textit{JumpFactor} > 1$

Since the jump factor would have a bigger impact on markets in the early days of the protocol, allowing the first *err* to have too much impact on the jump factor would complicate parameter tuning.

**Recommendation:** Consider to set the first `err` at zero and adjust the parameters accordingly.

```
- if (marketIrm[id].prevBorrowRate == 0) return (err, INITIAL_RATE, INITIAL_RATE);
+ if (marketIrm[id].prevBorrowRate == 0) return (0, INITIAL_RATE, INITIAL_RATE);
```

We can also rewrite the JumpFactor to always depend on `err` instead of `errDelta`. We would have the same effect with less complexity. The only difference is that the `errDelta` would not be clipped by `MIN_RATE`, or `MAX_RATE` that was discussed here

```
jumpMultiplier = MathLib.wExp(errwMulDown(int256(LN_JUMP_FACTOR)));
```

**Morpho**

**Cantina:**

## 3.4 Informational

### 3.4.1 Check invariants and add unit tests for `wExp(x)`

**Severity:** Informational

**Context:**

- MathLib.sol#L23-L46

**Description/Recommendation:**

Check invariants and add unit tests for `wExp(x)`:

- add unit test to make sure `wExp(q * LN2_INT) == WAD_INT << q` for non-negative `q`.
- add unit test to verify `wMulDown(wExp(x), wExp(-x)` is approximately `WAD`:

$$f(x)f(-x) = 1$$

From shared internal documents regarding the jump factor:

> This way, one cannot manipulate the rate by supplying a huge amount and withdrawing right after.

- prove and add unit test to verify `wExp(x) >= 0`.
- check whether `max(0, WAD + x) <= wExp(x)`

**Morpho**

**Cantina:**

### 3.4.2 Safe upper-bound for `SPEED_FACTOR` needs to be documented

**Severity:** Informational

**Context:**

- SpeedJumpIrm.sol#L131-L135
- Morpho.sol#L456-L459

**Description:** `SPEED_FACTOR` ( $k_P$ ) is used in the calculation of `linearVariation`:

$$L \approx k_P \cdot \mathtt{err} \cdot \Delta t / 10^{18}$$

where:

| parameter | description |
|---|---|
| Δt | `elapsed` |
| L | `linearVariation` |
| $k_P$ | `SPEED_FACTOR` |
| `err` | `err` |

- `elapsed` would be at most `type(uint64).max` (since timestamps are accounted for for example in `geth` as `uint64`)
- `err` is bounded by $10^{18}$

So for `speed * int256(elapsed)` to be less than `type(int256).max` we possible upper bound for `SPEED_FACTOR` would be:

$$2^{191} < \frac{2^{255} - 1}{(2^{64} - 1)} < 2^{192}$$

so for example assigning `uint184` to `SPEED_FACTOR` or making sure `SPEED_FACTOR` is no more than $2^{191}$ in the constructor would suffice.

The NatSpec comments mention:

> A typical value for the SPEED_FACTOR would be 10 ethers / 365 days.

The value mentioned is way lower than the suggested upper bound. But It would be great to document and have the check implemented.

**MathLib.wExp**

Regarding the approximate exponentiation the accepted upperbound is around `135 ethers` for `linearVariation` which makes the suggested upper bound above:

$$\frac{135 \cdot 10^{18}}{(2^{64}-1)} \approx 7.318364664277 \cdots$$

which means the accepted range of values for `SPEED_FACTOR` is really small, unless one has an expected estimated upper bound for the elapsed time and also one needs to make sure the errors are also bounded enough and not so close to the `WAD`.

So, `SPEED_FACTOR`, elapsed time and error need to be controlled.

**Recommendation:** Safe upper-bound for `SPEED_FACTOR` needs to be documented. Funds would be frozen forever if parameters are not set carefully.

**Morpho** Yes good point, I agree that it needs to be documented.

A few remarks to have in mind:

- `err` is bounded by `1e18` (your comment about being bounded by `2e18` applies to `errDelta`). We don't want to put bounds on `err`, to let the market evolve freely, so this seems like a good approximation.
- $2^{64}$ for the time elapsed seems unrealistic, it corresponds to 585 billion years.
- making sure that markets are updated every year should not be an issue, and this bound corresponds to reasonable values of the speed factor, where the typical value of the speed factor you mention is about 13 times lower

**Cantina:** If `err` gets too close to $10^{18}$ we would need to have:

$$k_P \cdot \Delta t \leq 135 \cdot 10^{18}$$

**Morpho** So bounding elapsed to a year, this gives and upper bound of `135 ethers per year` which is enough for the typical value of `10 ethers per year`

**Cantina** Also safe bounds would need to be estimated when one calculates the interest in `Morpho`:

```
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();
```

`interest` would need to fit in `uint128`.

$$\frac{A_B}{10^{18}} \cdot \left(x + \frac{x^2}{2! \cdot 10^{18}} + \frac{x^3}{3! \cdot 10^{36}}\right) \leq 2^{128} - 1$$

| parameter | description |
|---|---|
| $A_B$ | `totalBorrowAssets` |
| x | $\Delta t \bar{r}$ |
| $\Delta t$ | `elapsed` |
| $\bar{r}$ | `borrowRate` or `avgBorrowRate` |

Assuming $A_B$ is $10^{18}$, the maximum for x would be $1.2686160381663402351225217 \cdot 10^{7+18}$

The requirement can be even more strict as interests are accumulated into $A_B$ (and also $A_S$ the `totalSupplyAssets`):

$$\frac{A_B}{10^{18}} \cdot (10^{18} + x + \frac{x^2}{2! \cdot 10^{18}} + \frac{x^3}{3! \cdot 10^{36}}) \leq 2^{128} - 1$$

Above might also influence the value picked for `MAX_RATE` which is used to clip the returned average borrow rate.

### 3.4.3 Return early in `wExp(x)` when `x` is really small

**Severity:** Informational

**Context:**

- MathLib.sol#L28

**Description:** Based on the suggested bounds for x from issues/35

$$-8.352633645 \cdots 10^{58} \leq q \leq 195 \quad (\texttt{or} 196)$$

Similar to `remco`'s `FixedPointMathLib` one could return `0` earlier when `x` is below a certain threshold.

**Recommendation:** For simpler analysis, it would be best to return `0` for small values of `x` below a certain threshold instead of having a underflow revert requirement.

**Morpho**

**Cantina:**

### 3.4.4 Incorrect comments for `wExp(x)` and `errDelta`

**Severity:** Informational

**Context:**

- MathLib.sol#L27
- MathLib.sol#L35
- MathLib.sol#L39
- SpeedJumpIrm.sol#L125

**Description/Recommendation:**

- MathLib.sol#L27, `type(int256).min = -2**255`:

```diff
- // Revert if x < -(2**255-1) + (ln(2)/2).
+ // Revert if x < -2**255 + (ln(2)/2).
```

- MathLib.sol#L35

Let $\epsilon_0 =$

### 3.4.5 Natspec documentation issues: missed parameters, typos or suggested updates

**Severity:** Informational

**Context:**

- MathLib.sol
- UtilsLib.sol

**Description:** We have found different natspec documentation issues that include missing parameters, typos or in general suggestion to better improve them.

- MathLib.sol: some functions are missing the `@notice`, `@param` and `@return` natspec statement
- UtilsLib.sol: some functions are missing the `@notice`, `@param` and `@return` natspec statement

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.

**Morpho** We acknowledge this issue