# Morpho/Mystic Leverage Bundler Vulnerabilities Report

## Vulnerability #1: Inaccurate Collateral Calculation During Partial Position Closure Due to Interest Accounting

**HIGH SEVERITY**

### Summary

The MorphoLeverageBundler and MysticLeverageBundler contracts calculate collateral withdrawal amounts based on internal position tracking without accounting for accrued interest.

### Finding Description

The leverage bundlers track user positions using internal state variables:

```
totalBorrowsPerUser[pairKey][user]
totalCollateralsPerUser[pairKey][user]
```

When a user partially closes a position, the bundlers calculate how much collateral to withdraw using this formula:

```
uint256 collateralForRepayment = (totalCollateralsPerUser[pairKey][msg.sender]
```

This calculation assumes that the ratio between tracked debt and tracked collateral remains constant. However, in lending protocols, interest accrues on borrowed amounts over time, but this accrual is not reflected in the bundlers' internal tracking.

> **Example:**
> If a user initially borrows 100 ETH with 200 ETH collateral, and after some time the actual debt with interest is 110 ETH, the bundlers still track only the original 100 ETH. When the user wants to close 50% of their position:
>
> **Current calculation:** (200 ETH collateral * 50 ETH debt) / 100 ETH tracked debt = 100 ETH collateral to withdraw
>
> **Correct calculation:** (200 ETH collateral * 50 ETH debt) / 110 ETH actual debt = 90.9 ETH collateral to withdraw
>
> This discrepancy of 9.1 ETH excess collateral withdrawal will grow larger the longer the position is held and the more interest accrues.

## Impact Explanation

> 1. The withdrawal could reduce the health factor of the remaining position, bringing it closer to liquidation thresholds.
> 2. Users who intend to maintain a specific leverage ratio after partial closure will find their positions have higher leverage than expected.
> 3. The longer a position is held, the more interest accrues, and the greater the discrepancy becomes.

## Proof of Concept

A test contract added to the Bundler3 folder. Run the contract using the command:

```
forge test --mc MorphoLeverageInterestTest -vvvv
```

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

import "../lib/forge-std/src/Test.sol";
import "../src/calls/MorphoLeverageBundler.sol";
import {MarketParams, IMorpho} from "../lib/morpho-blue/src/interfaces/IMorpho.
import {IOracle} from "../lib/morpho-blue/src/interfaces/IOracle.sol";
import {MathRayLib} from "../src/libraries/MathRayLib.sol";
import {IERC20} from "../lib/openzeppelin-contracts/contracts/token/ERC20/IERC2

contract MorphoLeverageInterestTest is Test {
    using MathRayLib for uint256;

    // Define a basic test demonstrating the mathematical issue
    function testInterestAccountingDiscrepancy() public {
        // Initial position values
        uint256 initialBorrows = 100 ether;
        uint256 initialCollateral = 200 ether;

        // Record the initial ratio
        uint256 initialRatio = (initialCollateral * 1e18) / initialBorrows;

        // Simulate interest accrual (10% interest)
        uint256 interestAccrued = 10 ether; // 10% of 100
        uint256 currentBorrows = initialBorrows + interestAccrued; // Now 110 e

        // User wants to close 50% of their position
        uint256 positionClosePercent = 50;
        uint256 debtToClose = initialBorrows * positionClosePercent / 100; // 5

        // How the bundler calculates collateral:
        // It uses the proportion of debt being closed to the tracked debt
        uint256 bundlerCalculatedProportion = (debtToClose * 1e18) / initialBor
        uint256 bundlerWithdrawnCollateral = (initialCollateral * bundlerCalcul

        // How it should calculate with accrued interest:
        // Using proportion of debt being closed to the current debt with inter
        uint256 correctProportion = (debtToClose * 1e18) / currentBorrows; // 5
        uint256 correctWithdrawnCollateral = (initialCollateral * correctPropor

        // The bundler will withdraw too much collateral
        uint256 excessCollateralWithdrawn = bundlerWithdrawnCollateral - correc

        // Output for clarity
        console.log("Initial borrows:", initialBorrows / 1 ether, "ETH");
        console.log("Initial collateral:", initialCollateral / 1 ether, "ETH");
        console.log("Interest accrued:", interestAccrued / 1 ether, "ETH");
        console.log("Current borrows with interest:", currentBorrows / 1 ether,
        console.log("Debt to close (50%):", debtToClose / 1 ether, "ETH");
        console.log("Bundler withdrawn collateral:", bundlerWithdrawnCollateral
        console.log("Correct withdrawn collateral:", correctWithdrawnCollateral
        console.log("Excess collateral withdrawn:", excessCollateralWithdrawn /

        // Assert that there is a discrepancy due to interest accounting
        assertGt(excessCollateralWithdrawn, 0, "Interest accounting should caus

        // The formula in both bundlers:
        // collateralForRepayment = (totalCollateralsPerUser[pairKey][msg.sende
        // totalBorrowsPerUser[pairKey][msg.sender];

        // Demonstrate how the bundler calculates vs how it should calculate:
        console.log("\nDemonstrating bundler calculation:");
        console.log("collateralForRepayment = (totalCollateralsPerUser * debtTo
        console.log("collateralForRepayment = (200 * 50) / 100 = 100 ETH");

        console.log("\nCorrect calculation with interest:");
        console.log("collateralForRepayment = (totalCollateralsPerUser * debtTo
        console.log("collateralForRepayment = (200 * 50) / 110 = 90.9 ETH");

        // This confirms the issue: when closing positions partially, the bundl
        // interest accrual, resulting in incorrect collateral withdrawal propo
    }
}
```

## Test Output

```
Logs: Initial borrows: 100 ETH Initial collateral: 200 ETH Interest accrued:
10 ETH Current borrows with interest: 110 ETH Debt to close (50%): 50 ETH
Bundler withdrawn collateral: 100 ETH Correct withdrawn collateral: 90 ETH
Excess collateral withdrawn: 9 ETH Demonstrating bundler calculation:
collateralForRepayment = (totalCollateralsPerUser * debtToClose) /
totalBorrowsPerUser collateralForRepayment = (200 * 50) / 100 = 100 ETH
Correct calculation with interest: collateralForRepayment =
(totalCollateralsPerUser * debtToClose) / currentBorrowsWithInterest
collateralForRepayment = (200 * 50) / 110 = 90.9 ETH
```

## Recommendation

1. **Modify the collateral calculation formula to use the current debt from the protocol:**

```
// Get current debt from the protocol
uint256 currentDebtWithInterest = morpho.expectedBorrowAssets(marketParams
// Calculate collateral proportionally to current debt
uint256 collateralForRepayment = (totalCollateralsPerUser[pairKey][msg.ser
```

2. Alternatively, implement a mechanism to periodically sync the internal position tracking with the actual protocol state to account for accrued interest.

## Vulnerability #2: Inadequate Slippage Protection Due to Lack of Price Re-evaluation in bundler3.multicall

INFORMATIONAL

### Summary

While price deviation checks are performed during bundle creation, there are no corresponding checks during bundle execution.

### Finding Description

The bundlers uses a price deviation check in the getMarketPairKey function that verifies the price ratio between collateral and loan assets is within a 5% deviation range:

```
function getMarketPairKey(MarketParams calldata marketParams) public view retur
    // ...
    uint256 ratio = (collateralPrice * SLIPPAGE_SCALE) / borrowPrice;
    require(ratio <= SLIPPAGE_SCALE + 500 && ratio >= SLIPPAGE_SCALE - 500, "Pr
        // ...
    }
```

This check is only done during bundle creation when functions like `createOpenLeverageBundle` are called. However, there is no mechanism to re-validate these prices when the bundle is actually executed via `bundler3.multicall(bundle)`.

If prices change, users will receive positions with different risk profiles than expected.

## Impact Explanation

1. It will lead to increased liquidation risk
2. It causes unpredictable position parameters
3. The lack of slippage protection increases the chance of sandwich attacks or price manipulation, especially for large positions

## Proof of Concept

A test contract added to the Bundler test folder. Run with the command:

```
forge test --mc SlippageProtectionTest -vvvv
```

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

import "../lib/forge-std/src/Test.sol";
import "../src/libraries/ErrorsLib.sol";
import "./helpers/mocks/OracleMock.sol";
import "./helpers/mocks/ERC20Mock.sol";

contract SlippageProtectionTest is Test {
    OracleMock public collateralOracle;
    ERC20Mock public collateralToken;
    ERC20Mock public borrowToken;
    address constant USER = address(0x1);
    uint256 constant SLIPPAGE_SCALE = 10000;

    // This simulates a leverage position
    struct Position {
        uint256 collateralAmount; // Amount of collateral token
        uint256 borrowAmount; // Amount of borrow token
        uint256 targetLeverage; // Target leverage in basis points (e.g., 20000
        uint256 initialPrice; // Initial price when position was created
    }

    function setUp() public {
        // Create tokens
        collateralToken = new ERC20Mock("Collateral", "COL");
        borrowToken = new ERC20Mock("Borrow", "BOR");

        // Setup oracle
        collateralOracle = new OracleMock(1e18); // Initial price 1.0
    }

    // Create a leverage position
    function createPosition(uint256 initialCollateral, uint256 targetLeverage)
        // For a 2x leverage position with 1 ETH initial collateral:
        // - We have 1 ETH of our own capital
        // - We borrow 1 ETH worth of tokens
        // - Total position value is 2 ETH (2x leverage)

        uint256 totalPosition = initialCollateral * targetLeverage / SLIPPAGE_S
        uint256 borrowAmount = totalPosition - initialCollateral;

        return Position({
            collateralAmount: totalPosition, // Total collateral after leverage
            borrowAmount: borrowAmount, // Amount borrowed
            targetLeverage: targetLeverage, // Target leverage
            initialPrice: collateralOracle.price() // Current price at creation
        });
    }

    // Calculate current leverage of a position
    function calculateLeverage(Position memory position) public view returns (u
        // Current price of collateral
        uint256 currentPrice = collateralOracle.price();

        // Current value of the collateral in USD
        uint256 collateralValue = position.collateralAmount * currentPrice / 1e

        // Current value of the debt in USD (assuming 1:1 for simplicity)
        uint256 debtValue = position.borrowAmount;

        // Equity = Collateral - Debt
        uint256 equity = collateralValue - debtValue;
        // Leverage = Collateral / Equity
        return collateralValue * SLIPPAGE_SCALE / equity;
    }

    function testInadequateSlippageProtection() public {
        // Create a 2x leverage position with 1 ETH initial collateral
        Position memory position = createPosition(1 ether, 20000);

        console.log("Position created:");
        console.log("- Initial price:", position.initialPrice);
        console.log("- Collateral amount:", position.collateralAmount);
        console.log("- Borrow amount:", position.borrowAmount);
        console.log("- Target leverage:", position.targetLeverage);

        // Verify current leverage matches target
        uint256 initialLeverage = calculateLeverage(position);
        console.log("Initial leverage:", initialLeverage);
        assertApproxEqAbs(initialLeverage, position.targetLeverage, 10, "Initia

        // Now drop the collateral price by 10%
        collateralOracle.setPrice(position.initialPrice * 90 / 100);
        console.log("Collateral price dropped by 10% to:", collateralOracle.pri

        // Calculate new leverage
        uint256 newLeverage = calculateLeverage(position);
        console.log("New leverage after price drop:", newLeverage);

        // Leverage should increase when collateral price drops
        assertGt(newLeverage, position.targetLeverage, "Leverage should increas
        uint256 leverageIncrease = newLeverage - position.targetLeverage;
        console.log("Leverage increased by:", leverageIncrease, "basis points")

        // Now increase the price by 20% from the lower level
        collateralOracle.setPrice(position.initialPrice * 110 / 100);
        console.log("Collateral price increased to:", collateralOracle.price())

        // Calculate new leverage
        newLeverage = calculateLeverage(position);
        console.log("New leverage after price increase:", newLeverage);

        // Leverage should decrease when collateral price rises
        assertLt(newLeverage, position.targetLeverage, "Leverage should decreas
        uint256 leverageDecrease = position.targetLeverage - newLeverage;
        console.log("Leverage decreased by:", leverageDecrease, "basis points")
    }
}
```

### Test Output

```
Logs: Position created: - Initial price: 1000000000000000000 - Collateral
amount: 2000000000000000000 - Borrow amount: 1000000000000000000 - Target
leverage: 20000 Initial leverage: 20000 Collateral price dropped by 10% to:
900000000000000000 New leverage after price drop: 22500 Leverage increased
by: 2500 basis points Collateral price increased to: 1100000000000000000 New
leverage after price increase: 18333 Leverage decreased by: 1667 basis
points
```

## Recommendation

1. **Add Slippage Parameters to Bundle Creation:**

```
function createOpenLeverageBundle(
    MarketParams calldata marketParams,
    address inputAsset,
    uint256 initialCollateralAmount,
    uint256 targetLeverage,
    uint256 maxAcceptableLeverage,
    uint256 minAcceptableLeverage,
    uint256 slippageTolerance
) external returns (Call[] memory bundle)
```

2. **Add a final validation call at the end of each bundle that checks:**

```
// Example implementation
function validatePosition(
    bytes32 pairKey,
    address user,
    uint256 minAcceptableLeverage,
    uint256 maxAcceptableLeverage
) external view {
    uint256 collateral = totalCollateralsPerUser[pairKey][user];
    uint256 borrowed = totalBorrowsPerUser[pairKey][user];
    uint256 currentLeverage = (collateral * SLIPPAGE_SCALE) / (collateral

    require(currentLeverage >= minAcceptableLeverage, "Leverage too low");
    require(currentLeverage <= maxAcceptableLeverage, "Leverage too high")
}
```

3. **Add an expiration timestamp to bundles, after which they cannot be executed:**

```
function createOpenLeverageBundle(
    // existing parameters
    uint256 deadline
)
```

4. **Replace the current oracle implementation with Chainlink price feeds that provide:**

```
// Example implementation
function verifyPrice(address asset) internal view returns (uint256) {
    (, int256 price, uint256 startedAt, uint256 updatedAt, ) =
        AggregatorV3Interface(priceFeeds[asset]).latestRoundData();

    require(updatedAt >= block.timestamp - maxPriceAge, "Oracle data too 
    require(startedAt != 0, "Round not complete");
    return uint256(price);
}
```