CANTINA

Metamorpho and Periphery

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-irm. The participants identified a total of **292** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 13
- Low Risk: 113
- Gas Optimizations: 10
- Informational: 156

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

3 Findings

3.1 Medium Risk

3.1.1 Missing important check if `repay` and `redeem` functions in `compoundv2migrationbundler.sol` are successful

Submitted by tsvetanovv, also found by T1MOH, Christoph Michel and Bauchibred

Severity: Medium Risk

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

Description: In `CompoundV2MigrationBundler.sol` we have `compoundV2Repay()` and `compoundV2Redeem()`:

```
function compoundV2Repay(address cToken, uint256 amount) external payable protected {
  if (cToken == C_ETH) {
    amount = Math.min(amount, address(this).balance);

    require(amount != 0, ErrorsLib.ZERO_AMOUNT);

    ICEth(C_ETH).repayBorrowBehalf{value: amount}(initiator());
  } else {
    address underlying = ICToken(cToken).underlying();

    if (amount != type(uint256).max) amount = Math.min(amount, ERC20(underlying).balanceOf(address(this)));

    require(amount != 0, ErrorsLib.ZERO_AMOUNT);

    _approveMaxTo(underlying, cToken);

    ICToken(cToken).repayBorrowBehalf(initiator(), amount);
  }
}
```

```
function compoundV2Redeem(address cToken, uint256 amount) external payable protected {
  amount = Math.min(amount, ERC20(cToken).balanceOf(address(this)));

  require(amount != 0, ErrorsLib.ZERO_AMOUNT);

  ICToken(cToken).redeem(amount); //@audit - not check return value
}
```

- `compoundV2Repay()`: Repays a borrow position on Compound V2.
- `compoundV2Redeem()`: Redeems assets from a cToken position on Compound V2. It calculates the redeemable amount based on the bundler's cToken balance and then calls the `redeem` function of the cToken contract.

The problem here is that `compoundV2Repay()` calls `repayBorrowBehalf` and `compoundV2Redeem()` calls `redeem()`. These functions will not rollback the transaction on failure but return 0 or `Error Code`. You can see the docs for repayBorrowBehalf and redeem.

One small disclaimer is that with `repayBorrowBehalf()` at `C_ETH` the function will revert if the transaction fails:

> `RETURN`: No return, reverts on error.

Recommendation: You should always make sure that these functions will not fail without reverting. Therefore it is mandatory to check the returned value and revert the transaction if unsuccessful.

3.1.2 Broken slippage check in `erc4626.erc4626mint`

Submitted by Christoph Michel, also found by T1MOH

Severity: Medium Risk

Context: ERC4626Bundler.sol#L36

Description: A user specifies `shares` and a `maxAssets` slippage parameter for the `erc4626Mint` function. However, the slippage parameter is not directly applied to the `shares` *parameter*, it is applied to the smaller shares value of:

```
shares = Math.min(shares, IERC4626(vault).maxMint(receiver));
```

This leads to the user potentially accepting a worse share price for the mint than they specified in the function.

Example: A user wants to mint 1000 shares at a share price of 1.0 and specifies `shares = 1000`, `maxAssets = 1000`. Imagine `maxMint` returning only 1 share and the share price suddenly being `1000.0`. The user now pays 1000 assets for a single share and the slippage check still passes.

Recommendation: Revert if the `shares` are less than `maxMint`, or adjust the `maxAssets` proportional to `maxMint/initialDesiredShares` to enforce the same share price.

3.1.3 Cannot use `uint256.max` to repay the balance of `aavev2migrationbundler`

Submitted by Christoph Michel, also found by J4X98

Severity: Medium Risk

Context: AaveV2MigrationBundler.sol#L40

Description: The `aaveV2Repay`'s natspec states the following for the `amount` parameter:

> The amount of `asset` to repay. Pass `type(uint256).max` to repay the bundler's `asset` balance.

Passing `type(uint256).max` to repay the bundler's `asset` balance does not work. Note that if `amount == type(uint256).max`, this value is *not* adjusted and just forwarded to the `AAVE_V2_POOL.repay` call.

Aave will execute this code:

```
uint256 paybackAmount =
  interestRateMode == DataTypes.InterestRateMode.STABLE ? stableDebt : variableDebt;

// not executed as params.amount = type(uint256).max
// meaning paybackAmount will stay at entire `variableDebt`
if (amount < paybackAmount) {
  paybackAmount = amount;
}
```

This caps the amount to be repaid to the **entire borrow balance**, not to the bundler's balance. The call will revert as it tries to repay the entire borrow balance with the bundler's balance.

Example: A user tries to migrate part of their position by trying to repay half of their borrow balance of 1000 assets. The first action in the bundle redeems shares from another protocol to receive the desired repay amount of roughly 500 assets, and the second action is to repay by setting `assets = type(uint256).max` to "repay the bundler's `asset` balance", as defined by the natspec. The batch will revert as Aave V2 will try to repay the entire borrow balance of 1000 but the bundler only has 500 assets.

Recommendation: Consider always capping the `amount` to the bundler's balance. Trying to repay more will never work, and the protocol itself will already cap it to the entire borrow balance:

```
- if (amount != type(uint256).max) amount = Math.min(amount, ERC20(asset).balanceOf(address(this)));
+ amount = Math.min(amount, ERC20(asset).balanceOf(address(this)));
```

3.1.4 Cannot use `uint256.max` to repay the balance of `aavev3migrationbundler`

Submitted by Christoph Michel, also found by J4X98

Severity: Medium Risk

Context: AaveV3MigrationBundler.sol#L37

Description: The `aaveV3Repay`'s natspec states the following for the `amount` parameter:

> The amount of `asset` to repay. Pass `type(uint256).max` to repay the bundler's `asset` balance.

Passing `type(uint256).max` to repay the bundler's `asset` balance does not work. Note that if `amount == type(uint256).max`, this value is *not* adjusted and just forwarded to the `AAVE_V3_POOL.repay` call.

AaveV3 will execute this code:

```
uint256 paybackAmount = params.interestRateMode == DataTypes.InterestRateMode.STABLE
  ? stableDebt
  : variableDebt;

// note that useATokens is false for repay and this is not executed
if (params.useATokens && params.amount == type(uint256).max) {
  params.amount = IAToken(reserveCache.aTokenAddress).balanceOf(msg.sender);
}

// not executed either as params.amount = type(uint256).max
// meaning paybackAmount will stay at entire `variableDebt`
if (params.amount < paybackAmount) {
  paybackAmount = params.amount;
}
```

Note that `useATokens` is `false` for repays. This caps the amount to be repaid to the **entire borrow balance**, not to the bundler's balance. The call will revert as it tries to repay the entire borrow balance with the bundler's balance.

Example: A user tries to migrate part of their position by trying to repay half of their borrow balance of 1000 assets. The first action in the bundle redeems shares from another protocol to receive the desired repay amount of roughly 500 assets, and the second action is to repay by setting `assets = type(uint256).max` to "repay the bundler's `asset` balance", as defined by the natspec. The batch will revert as Aave V3 will try to repay the entire borrow balance of 1000 but the bundler only has 500 assets.

Recommendation: Consider always capping the `amount` to the bundler's balance. Trying to repay more will never work, and the protocol itself will already cap it to the entire borrow balance:

```
- if (amount != type(uint256).max) amount = Math.min(amount, ERC20(asset).balanceOf(address(this)));
+ amount = Math.min(amount, ERC20(asset).balanceOf(address(this)));
```

3.1.5 Cannot use `uint256.max` to repay the balance of `aavev3optimizermigrationbundler`

Submitted by Christoph Michel, also found by J4X98

Severity: Medium Risk

Context: AaveV3OptimizerMigrationBundler.sol#L37

Description: The `aaveV3OptimizerRepay`'s natspec states the following for the `amount` parameter:

> The amount of `underlying` to repay. Pass `type(uint256).max` to repay the bundler's `underlying`

Passing `type(uint256).max` to repay the bundler's `underlying` balance does not work. Note that if `amount == type(uint256).max`, this value is *not* adjusted and just forwarded to the `AAVE_V3_OPTIMIZER.repay` call.

AaveV3Optimizier will forward the calls to Morpho's PositionManager which executes:

```
// AaveV3Optimizer
function _repay(address underlying, uint256 amount, address from, address onBehalf) internal returns (uint256)
{
    bytes memory returnData = _positionsManager.functionDelegateCall(
        abi.encodeCall(IPositionsManager.repayLogic, (underlying, amount, from, onBehalf))
    );

    return (abi.decode(returnData, (uint256)));
}

// PositionManager
function repayLogic(address underlying, uint256 amount, address repayer, address onBehalf)
external
returns (uint256)
{
  amount = Math.min(_getUserBorrowBalanceFromIndexes(underlying, onBehalf, indexes), amount);
}
```

This caps the amount to be repaid to the **entire borrow balance**, not to the bundler's balance. The call will revert as it tries to repay the entire borrow balance with the bundler's balance.

Example: A user tries to migrate part of their position by trying to repay half of their borrow balance of 1000 assets. The first action in the bundle redeems shares from another protocol to receive the desired repay amount of roughly 500 assets, and the second action is to repay by setting `assets = type(uint256).max` to "repay the bundler's `asset` balance", as defined by the natspec. The batch will revert as AaveV3Optimizer will try to repay the entire borrow balance of 1000 but the bundler only has 500 assets.

Recommendation: Consider always capping the `amount` to the bundler's balance. Trying to repay more will never work, and the protocol itself will already cap it to the entire borrow balance:

```
- if (amount != type(uint256).max) amount = Math.min(amount, ERC20(underlying).balanceOf(address(this)));
+ amount = Math.min(amount, ERC20(underlying).balanceOf(address(this)));
```

3.1.6 Cannot use `uint256.max` to repay the balance of `compoundv2migrationbundler`

Submitted by Christoph Michel

Severity: Medium Risk

Context: CompoundV2MigrationBundler.sol#L40

Description: The `compoundV2Repay`'s natspec states the following for the `amount` parameter:

> The amount of `cToken` to repay. Pass `type(uint256).max` to repay all (except for cETH).

It's ambiguous what "all" refers to, the bundler's token balance, or the borrow balance. We assume the bundler's balance as that's the approach taken by the other bundlers. Passing `type(uint256).max` to repay the bundler's `cToken` balance does not work. Note that if `amount == type(uint256).max`, this value is *not* adjusted and just forwarded to the `ICToken(cToken).repayBorrowBehalf` call.

CompoundV2 will execute this code:

```
/* We fetch the amount the borrower owes, with accumulated interest */
uint accountBorrowsPrev = borrowBalanceStoredInternal(borrower);

/* If repayAmount == -1, repayAmount = accountBorrows */
uint repayAmountFinal = repayAmount == type(uint).max ? accountBorrowsPrev : repayAmount;
```

This caps the amount to be repaid to the **entire borrow balance**, not to the bundler's balance. The call will revert as it tries to repay the entire borrow balance with the bundler's balance.

Example: A user tries to migrate part of their position by trying to repay half of their borrow balance of 1000 assets. The first action in the bundle redeems shares from another protocol to receive the desired repay amount of roughly 500 assets, and the second action is to repay by setting `assets = type(uint256).max` to "repay the bundler's `asset` balance", as defined by the natspec. The batch will revert as Aave V2 will try to repay the entire borrow balance of 1000 but the bundler only has 500 assets.

Recommendation: Consider always capping the `amount` to the bundler's balance. Trying to repay more will never work, and the protocol itself will already cap it to the entire borrow balance:

```
- if (amount != type(uint256).max) amount = Math.min(amount, ERC20(underlying).balanceOf(address(this)));
+ amount = Math.min(amount, ERC20(underlying).balanceOf(address(this)));
```

3.1.7 `max_rate_at_target` is given as apr but morpho blue uses it as apy

Submitted by Christoph Michel, also found by darkbit

Severity: Medium Risk

Context: ConstantsLib.sol#L9

Description: Morpho Blue is compounding the interest, see Morpho.sol#L477. The `MAX_RATE_AT_TARGET` specified here is given as a value that would result in an `1e7`% APR (annual percentage rate, non compounding). As Morpho Blue is compounding, this value must be given such that it results in an **APY** of `1e7`%. We need to compute it the same way as Morpho does as `e^{nx} - 1 = 0.01e9` where n = 365 days. Solving for n*x = 16.1181. Therefore, `x = MAX_RATE_AT_TARGET = 16.1181 ether / 365 days`. The current value of `1e7 ether / 365 days` would lead to an actual Morpho Blue APY (yearly compounded interest rate) of e^1e7 = 6.592232534618439×10^4342944 which is a number with 4 million digits.

If an IRM uses this rate, the market would become dysfunctional because of the Morpho computation overflowing already after `exp(n*x) - 1 = 560` seconds. Markets using this IRM will revert in the accrual code less than 10 minutes after the last market accrual. Users will be unable to withdraw their funds as `withdraw` and almost all other functions always perform a market accrual first.

The likelihood is high because the max borrow rate can be reached by any IRM using the adaptive rebasing mechanism. Also, IRM borrow rates are supposed to be in range of not reverting by this code and not checked by a human when white-listing. The impact is high as all user funds in the pool will be locked due to the revert during accruing interest.

Recommendation: Consider using the correct APY values mentioned above.

3.1.8 `min_rate_at_target` is given as apr but morpho blue uses it as apy

Submitted by Christoph Michel

Severity: Medium Risk

Context: ConstantsLib.sol#L12

Description: Morpho Blue is compounding the interest, see Morpho.sol#L477. The `MIN_RATE_AT_TARGET` specified here is given as a value that would result in an 0.1% APR (annual percentage rate, non compounding). As Morpho Blue is compounding, this value must be given such that it results in an **APY** of 0.1%. We need to compute it the same way as Morpho as `e^{nx} - 1 = 0.001` where n = 365 days. Solving for n*x 0.0009995. Therefore, `x = MAX_RATE_AT_TARGET = 0.0009995 ether / 365 days`. The current value of `0.001 ether / 365 days` would lead to an actual Morpho blue APY (yearly compounded interest rate) of e^{0.001} = 0.0010005 = 0.10005%.

The desired lower bound of `0.1`% cannot be reached.

Recommendation: Consider using the correct APY values mentioned above.

3.1.9 Wrong rounding direction in `_maxdeposit`

Submitted by Christoph Michel

Severity: Medium Risk

Context: MetaMorpho.sol#L651

Description: The `_maxDeposit` should over-estimate the returned `totalSuppliable` assets, as otherwise, when over-estimating, a `deposit` using this `_maxDeposit` value might revert. This means `totalSuppliable` should round down, which means the `supplyCap.zeroFloorSub(supplyAssets)` term should round down, which means `supplyAssets` should round up. However, `supplyAssets = MORPHO.expectedSupplyAssets(_marketParams(id), address(this))` is rounding down.

Recommendation: Consider simulating `MORPHO.expectedSupplyAssets(_marketParams(id), address(this))` but by rounding up.

3.1.10 Wrong rounding direction for `tosupply` in `_supplymorpho`

Submitted by Christoph Michel

Severity: Medium Risk

Context: MetaMorpho.sol#L807

Description: A property that a `_supplyMorpho` call should have is that immediately after the deposit, the supplied assets on the market from Metamorpho do not exceed Metamorpho's `supplyCap`. This means the `toSupply` computation should under-estimate the assets:

```
(uint256 supplyAssets,,) = _accruedSupplyBalance(marketParams, id);
 uint256 toSupply = UtilsLib.min(supplyCap.zeroFloorSub(supplyAssets), assets);
```

This means the `supplyCap.zeroFloorSub(supplyAssets)` should round down, i.e., the `supplyAssets` should round up. However, `supplyAssets` are rounded down. A deposit on Metamorpho can end up immediately breaking its `supplyCap`s.

Recommendation: Consider simulating `_accruedSupplyBalance()` but by rounding up the final computation:

```
assets = shares.toAssetsUp(market.totalSupplyAssets, market.totalSupplyShares);
```

3.1.11 Compoundv3migrationbundler is uncapable of withdrawing user's collateral from the compound's instances

Submitted by 0xStalin

Severity: Medium Risk

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

When the `CompoundV3MigrationBundler.compoundV3WithdrawFrom()` function is called to attempt to withdraw the user's collateral, the whole execution of the multicall transaction will be reverted because a runtime compilation error will occur when the returned value from the `Compound.userCollateral()` function (a `struct`) is tried to be assigned to the balance variable (uint256) [See the Proof of Concept section for more details]

Proof of Concept:

- The CompoundV3MigrationBundler intends to allow users to Migrate positions from CompoundV3 to Morpho Blue, one of the functions that users can use to manage their compound's position and move their assets to Morpho Blue is the `compoundV3WithdrawFrom()` function, which allows users to withdraw their assets from Compound, this function allows users to withdraw the user's collateral that was deposited in Compound, the problem is that the function `ICompoundV3(instance).userCollateral()` returns a `struct`, instead of a `uint`, because of the returned value is a struct, when the value is tried to be assigned to the `balance` variable, a `Type Error` will occur and will revert all the multicall execution, causing users to not being able to migrate their assets from compound to morpho, and also, users would waste their gas paying for a transaction that always reverts.
- Let's do a walkthrough of the contracts to spot the exact issue:
  - CompoundV3MigrationBundler.sol

```
  function compoundV3WithdrawFrom(address instance, address asset, uint256 amount) external payable
↪   protected {
      address _initiator = initiator();
      uint256 balance = asset == ICompoundV3(instance).baseToken()
          //@audit-ok => Compound.balanceOf() returns a uint256
          ? ICompoundV3(instance).balanceOf(_initiator)

          //@audit-issue => Compund.userCollateral() of the returns a struct, not a uint.
          //@audit-issue => When trying to assign the returned value to the `balance` variable which is
↪   an uint256 it will throw an error about incompatible types
          : ICompoundV3(instance).userCollateral(_initiator, asset);

      amount = Math.min(amount, balance);

      require(amount != 0, ErrorsLib.ZERO_AMOUNT);

      ICompoundV3(instance).withdrawFrom(_initiator, address(this), asset, amount);
  }
```

- By looking at the code of the CompoundV3 Protocol, we realize that the call to `userColleral()` actually calls a public mapping defined in the CometStorage.sol (which is a contract that main contract inherits from), and this mapping ends up returning a `struct`
  - CompoundV3 Protocol, CometStorage.sol

```
  contract CometStorage {
    // ...
    //@audit-info => This is the returned Struct when userCollateral() is called!
    struct UserCollateral {
      uint128 balance;
      uint128 _reserved;
    }
    // ...

    //@audit-info => This is the public mapping that is called from the
↪   CompoundV3MigrationBundler::compoundV3WithdrawFrom()
    mapping(address => mapping(address => UserCollateral)) public userCollateral;
  }
```

- As we've just seen in the previous walkthrough, as a result of not decoding the received struct and just trying to assign it directly to a variable of type uin256, the whole execution will be blown up. All changes will be reverted, causing users to lose all the gas that will be paid for the failed execution.

Recommendation: The mitigation for this issue is to make sure to decode the received `struct` and assign the value of the variable `balance`.

- CompoundV3MigrationBundler.sol

```
  function compoundV3WithdrawFrom(address instance, address asset, uint256 amount) external payable protected {
      address _initiator = initiator();
-     uint256 balance = asset == ICompoundV3(instance).baseToken()
-         ? ICompoundV3(instance).balanceOf(_initiator)
-         : ICompoundV3(instance).userCollateral(_initiator, asset);

+     uint256 balance;
+     if(asset == ICompoundV3(instance).baseToken()) {
+         balance = ICompoundV3(instance).balanceOf(_initiator);
+     } else {
+       //@audit-ok => Extract the value of the `balance` variable returned from the Compound Contract
+       (uint128 _balance, ) = ICompoundV3(instance).balanceOf(_initiator);
+       balance = uint256(_balance);
+     }

      amount = Math.min(amount, balance);

      require(amount != 0, ErrorsLib.ZERO_AMOUNT);

      ICompoundV3(instance).withdrawFrom(_initiator, address(this), asset, amount);
  }
```

3.1.12 Virtual supply shares steal interest

Submitted by Christoph Michel

Severity: Medium Risk

Context: MetaMorpho.sol#L590

Description: The virtual supply shares, that are not owned by anyone, implicitly earn interest as the shares-to-asset conversions used in `withdraw` involve the virtual assets.

```
function _convertToSharesWithTotals(
    uint256 assets,
    uint256 newTotalSupply,
    uint256 newTotalAssets,
    Math.Rounding rounding
) internal pure returns (uint256) {
    return assets.mulDiv(newTotalSupply + 10 ** _decimalsOffset(), newTotalAssets + 1, rounding);
}
```

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.

3.1.13 `ERC20WrapperBundler` calls are missing return value checks

Submitted by J4X98

Severity: Medium Risk

Context: ERC20WrapperBundler.sol#L38, ERC20WrapperBundler.sol#L54

Description: The Morpho Blue bundler offers users the capability to bundle calls, including interactions with contracts that implement the `ERC20Wrapper` interface. This functionality is facilitated through the `erc20WrapperDepositFor()` and `erc20WrapperWithdrawTo()` functions.

The current implementation presents a vulnerability as it neglects to check the boolean return value of the underlying `ERC20Wrapper` functions. While the OpenZeppelin implementation returns true for successful calls and reverts on errors, alternative implementations may simply return false in case of an unsuccessful call, allowing the call to proceed without reverting.

Given that the documentation specifies the assumption that the wrapper implements the `ERC20Wrapper` interface without explicitly detailing the OpenZeppelin functionality, variations in implementation are valid. Consequently, this issue may result in a scenario where a user sends tokens for wrapping to the contract, but the wrapper fails to transfer them out of the bundler (e.g., due to a blocklist) and returns false. Since the return value is not checked, the execution continues, leaving the user's tokens within the bundler and vulnerable to theft.

This issue can lead to a potential loss of user funds as a user will expect the bundler to revert in the case of any of the transferring functions failing, but instead the bundler will finish the execution. leaving tokens in the bundler. These tokens could then be stolen by anyone.

Proof of Concept: To simulate this issue I have implemented a simple ERC20Wrapper that returns false on an incorrect deposit instead of reverting.

```
import {IERC20} from "../../lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";

contract ERC20WrapperNotReverting {
    IERC20 private immutable _underlying;
    mapping(address => uint256) public balances;

    constructor(IERC20 underlyingToken) {
        _underlying = underlyingToken;
    }

    function underlying() public view returns (IERC20) {
        return _underlying;
    }

    /**
     * @dev Allow a user to deposit underlying tokens and mint the corresponding number of wrapped tokens.
     */
    function depositFor(address account, uint256 amount) public virtual returns (bool) {
        if(msg.sender != address(0x123456))
        {
            //simulating only dedicated users being allowed to call the function
            return false;
        }

        _underlying.transferFrom(msg.sender, address(this), amount);
        balances[account] += amount;
        return true;
    }
}
```

In the following proof of concept one can see the issue happen:

```
function testErc20WrapperDepositForNoRevert() public {
    //Deploy the non reverting mock
    ERC20WrapperNotReverting wrapper2 = new ERC20WrapperNotReverting(loanToken);

    bundle.push(_erc20WrapperDepositFor(address(wrapper2), 100));

    loanToken.setBalance(address(bundler), 100);

    vm.prank(RECEIVER);
    bundler.multicall(bundle);

    //Tokens are still left in the contract but it didn't revert
    assertEq(loanToken.balanceOf(address(bundler)), 100, "loan.balanceOf(bundler)");
    assertEq(loanWrapper.balanceOf(RECEIVER), 0, "loanWrapper.balanceOf(RECEIVER)");
}
```

The testcase can be run by:

1. Adding the `ERC20WrapperNotReverting` contract to the `morpho-blue-bundlers/src/mocks` folder.
2. Importing it using `import {ERC20WrapperNotReverting} from "../../src/mocks/ERC20WrapperNotReverting.so`
3. Adding the testcase to `morpho-blue-bundlers/test/forge/ERC20WrapperBundlerBundlerLocalTest`.
4. Run it using `forge test -vvvv --match-test "testErc20WrapperDepositForNoRevert"`.

Recommendation: Mitigate this issue by incorporating return value checks for the calls to the functions `ERC20Wrapper(wrapper).depositFor()` and `ERC20Wrapper(wrapper).withdrawTo()`. The recommended adjustments are as follows:

- Deposit:

```
bool success = ERC20Wrapper(wrapper).depositFor(initiator(), amount);
require(success, "Deposit was unsuccessfull");
```

- Withdrawal:

```
bool success = ERC20Wrapper(wrapper).withdrawTo(account, amount);
require(success, "Withdraw was unsuccessfull");
```