Morpho Blue Bundlers

Security Review

Cantina Managed review by:
Saw-mon-and-Natalie, Lead Security Researcher **Jonah1005**, Lead Security Researcher **StErMi**, Security Researcher

November 14, 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

[PROJECT DESCRIPTION HERE]

From Sep 28th - Oct 16th the Cantina team conducted a review of Morpho Blue Bundlers on commit hash cdf003...aef953. The team identified a total of **18** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 5
- Gas Optimizations: 1
- Informational: 11

3 Findings

3.1 Medium Risk

3.1.1 Caller could enable malicious contract to re-enter the Bundler flow

Severity: Medium Risk

Context: TransferBundler.sol#L31

Description: With the current implementation of the Morpho Bundler, there are multiple endpoints where a malicious contract could be allowed to re-enter the Bundler flow.

1) `TransferBundler.nativeTransfer` is a bundler action that allows the `initiator` to transfer `ETH` to a `recipient` via a low-level `.call` implemented in Morpho Blue `TransferHelper` library contract. The `recipient` contract could intercept the Bundler flow in the `receive` callback and inject itself into the Bundler flow.
2) There are multiple entry points where the user is allowed to specify arbitrary addresses to be executed. Malicious actors could deceive the caller and force them to execute such an attack. One of the many examples could be `ERC4626Bundler.erc4626Deposit(address vault, uint256 assets, address owner)` where the user is giving `asset` allowance to an **arbitrary** `vault` address and later executing `vault.deposit(assets, owner)`. A malicious actor, owner of the `vault` contract, could at that point inject itself into the Bundler flow.

Once the malicious contract is executed, it will be allowed to re-enter the Bundler flow and perform both "direct operation" (without entering a recursive multicall) or by performing another multicall (that would override the `initiator` parameter with the malicious address itself).

If the malicious contract executes a "direct" call, it is allowed to perform all those operations that leverage the fact that the `initiator` global variable has been already initiated with the **original** `msg.sender`. For example, if the `initiator` has already approved the Bundler with unlimited allowance, the malicious actor would be able to execute `TransferBundler.erc20TransferFrom(asset, amount)`

```
function erc20TransferFrom(address asset, uint256 amount) external payable {
    amount = Math.min(amount, ERC20(asset).balanceOf(initiator));

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

    ERC20(asset).safeTransferFrom(initiator, address(this), amount);
}
```

And then transfer all the amount received by the Bundler back to the malicious contract (or an arbitrary receiver). In general, all those operations that use the `initiator` in a "forced" way are all open to be exploited in this scenario.

Re-entering the multicall and so re-initializing the `initiator` global state variable could open up to different kinds of attacks, but the revert of the whole transactions depends mostly on which actions are executed as soon as the second multi-call is ended and the `initiator` is resetted to `address(0)`.

```
function multicall(bytes[] memory data) external payable {
    initiator = msg.sender;

    _multicall(data);

    delete initiator;
}
```

Recommendation: 1) Unless there is a very specific use case, Morpho should not allow the `initiator` to re-enter a recursive multicall transaction (that could end up anyway to revert as soon as the second one exits the multicall and the following actions are executed with `initiator = address(0)`). 2) Morpho should consider adding some specific checks to prevent any malicious actor to re-enter the Bundler flow and being able to execute any actions, in particular when the `initiator` has been already initialized with the original `msg.sender` value. 3) Morpho should anyway document all these possible worst-case scenarios when the original caller executes multicall operations that interact with **arbitrary** contracts that could re-enter in a malicious way the Bundler flow.

Cantina:

With the PRs https://github.com/morpho-labs/morpho-blue-bundlers/pull/284 and https://github.com/morpho-labs/morpho-blue-bundlers/pull/315 Morpho has implemented the first and third points of the recommendations. The second point has been documented (as a warning) in functions natspec where needed.

Cantina:

The PR https://github.com/morpho-org/morpho-blue-bundlers/pull/354 implements the second point of the reccomendations. Now, only the `initiator` (that has started the very first `multicall`) or `morpho` itself can re-enter the bundler flow (within a `multicall`)

3.2 Low Risk

3.2.1 Consider adding a slippage protection mechanism to the `MorphoBundler` contract

Severity: Low Risk

Context: MorphoBundler.sol

Description: Like it has been suggested for the `ERC4626Bundler` (see ERC4626Bundler does not offer any slippage/min output protection or could end up reverting the operation), also the `MorphoBundler` should offer a slippage protection mechanism when the user interacts with the underlying Morpho Blue protocol.

- `morphoSupply` could allow the user to specify `maxAssets` when `shares > 0` and revert if the value returned by `morpho.supply` (relative to the amount of assets used to mint `shares`) is greater than `maxAssets`
- `morphoSupply` could allow the user to specify `minShares` when `amount > 0` and reverts if the value returned by the `morpho.supply` (Relative to the shares minted used `amount` of assets) is lower than `minShares`
- `morphoBorrow` could allow the user to specify `minAssets` when `shares > 0` and revert if the value returned by `morpho.borrow` (relative to the amount of assets sent to the receiver) is lower than `minAssets`
- `morphoBorrow` could allow the user to specify `maxShares` when `amount > 0` and revert if the value returned by `morpho.borrow` (relative to the amount of shares minted to borrow `amount` of assets) is greater than `maxShares`
- `morphoRepay` could allow the user to specify `maxAssets` when `shares > 0` and revert if the value returned by `mopho.repay` (relative to the amount of loan tokens to repay to burn `shares` of debt) is greater than `maxAssets`
- `morphoRepay` could allow the user to specify `minShares` when `amount > 0` and revert if the value returned by `morpho.repay` (relative to the amount of debt shares burned by providing `amount` of assets) is lower than `minShares`
- `morphoWithdraw` could allow the user to specify `minAssets` when `shares > 0` and revert if the value returned by `mopho.withdraw` (relative to the `amount` of assets withdrawn by burning `shares) is lower than` minAssets'
- `morphoWithdraw` could allow the user to specify `maxShares` when `assets > 0` and revert if the value returned by `morpho.withdraw` (relative to the `shares` burned to withdraw `amount` tokens) is greater than `maxShares`
- `morphoLiquidate` could allow the user to specify `minSeizedCollateral` when `repaidShares > 0` and revert if the value returned by `morpho.liquidate` (relative to the amount of collateral received to repay `repaidShares` of debt) is lower than `minSeizedCollateral`
- `morphoLiquidate` could allow the user to specify `maxRepaidAssets` when `repaidShares > 0` or `seizedAssets > 0` and revert if the value returned by `morpho.liquidate` (relative to the amount of `loanToken` repaid to burn `repaidShares` shares of debt or to receive `seizedAssets` of collateral) is greater than `maxRepaidAssets`

Recommendation: Morpho should consider adding support to slippage protection mechanisms to the `MorphoBundler` bundler.

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/314

3.2.2 ERC4626Bundler does not offer any slippage/min output protection or could end up reverting the operation

Severity: Low Risk

Context: ERC4626Bundler.sol

Description: The ERC4626Bundler contract allows the caller to interact with a parametric ERC4626 vault and performs the mint, deposit, redeem and withdraw operation on the vault itself. An ERC4626 default implementation does not offer any kind of slippage protection mechanism and, as a consequence, the ERC4626Bundler bundler does not offer it either.

Based on the user's parameters, the vault state and, depending on the operation, the balance of the bundler itself, the ERC4626Bundler could end up operating in an even more conservative way, allowing the user to mint, deposit, redeem or withdraw less than the user has specified without any revert. There are also cases where the operation instead could revert because the user has not provided enough funds to cover the operation requirements.

`erc4626Mint(address vault, uint256 shares, address owner)`

In this case, the user specifies the specific amount of `shares` that he/she wants to be minted

If `shares > IERC4626(vault).maxMint(owner)`, the user will end up minting fewer shares than desired If `IERC4626(vault).previewMint(shares) > ERC20(asset).balanceOf(address(this))`, the transaction will revert because the number of assets owned by the Bundler would end up not being enough to execute the `vault.mint` operation. In this case, to avoid the revert, Morpho should re-calculate the amount of `shares` that will be minted given the `asset` balance of the Bundler.

`erc4626Deposit(address vault, uint256 assets, address owner)`

In this case, the user specifies the specific amount of `assets` he wants to deposit into the `vault`. The number of shares minted is limited by these factors:

1) `IERC4626(vault).maxDeposit(owner)`: the max amount of assets that this specific user can deposit
2) `ERC20(asset).balanceOf(address(this))`: the amount of assets that the user has transferred to the bundler

Depending on those factors, the user could end up minting fewer shares than expected.

In this case, we should also consider that the user would like to protect itself from a possible vault's slippage and specify a `minShares` amount that could be lower compared to the return value of `vault.deposit`

`erc4626Withdraw(address vault, uint256 assets, address receiver)`

In this case, the user specifies the specific amount of `assets` he wants to withdraw from the vault.

Depending on the returned value of `vault.maxWithdraw` the user could end up withdrawing less `assets` compared to what he/she has specified as the input parameter.

Depending on the vault slippage and if the caller has given infinite allowance to the Bundler, the caller could end up burning more shares than expected to withdraw `assets`. In this case, Morpho could offer a `maxShares` input parameter that would make the operation reverts if the number of shares burned is greater than such upper limit.

`erc4626Redeem(address vault, uint256 shares, address receiver)`

In this case, the user specifies the specific amount of `shares` that should be redeemed from the `vault` to withdraw some assets.

Depending on the returned value of `vault.maxRedeem` the user could end up redeeming fewer shares compared to what have been specified as an input parameter and, as a consequence, less underlying than expected (given the initial number of shares).

The amount of underlying withdrawn could also be limited by the vault's slippage. In this case, Morpho should consider offering a slippage protection `minAssets` that would revert the operation if the amount of assets withdrawn (by redeeming the shares) is less than what is anticipated.

Recommendation: Morpho should prevent the revert of `erc4626Mint` when `IERC4626(vault).previewMint(shares) > ERC20(asset).balanceOf(address(this))`. In this case, the amount of `shares` minted should be recalculated based on the effective balance of the Bundler.

Morpho should consider adding the following slippage protection parameters:

- `uint256 minShares` to the `erc4626Deposit` function
- `uint256 maxShares` to the `erc4626Withdraw` function
- `uint256 minAssets` to the `erc4626Redeem` function

Morpho should also consider these possible side effects:

If they mint/deposit/withdraw/redeem less than what the user have anticipated and the multicall transaction is depending on those exact amounts, the multicall transaction could end up reverting down the line because future operations has not enough funds to perform it.

If the multicall transaction does not revert down the line, Morpho could end up not respecting the user's initial intention, and he/she should be warned about this possibility. Because the vault's operation's result is lower compared to what the user has asked, it means that also the following actions results will be lower and different from what the user would have expected.

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/310

3.2.3 `CompoundV3MigrationBundler` should revert if the Compound v3 execution is not performing what the bundler is supposed to do

Severity: Low Risk

Context: CompoundV3MigrationBundler.sol#L26-L35, CompoundV3MigrationBundler.sol#L41-L43, CompoundV3MigrationBundler.sol#L50-L52

Description: The `CompoundV3MigrationBundler` should only allow the caller to repay debt and withdraw supplied tokens or collateral in order to later migrate those amounts to Morpho Blue markets.

Depending on the status of the caller position (on Compound) and the parameters passed to `ICompoundV3(instance).supplyTo`, `ICompoundV3(instance).withdraw` or `ICompoundV3(instance).withdrawFrom` the user could end up performing the opposite operation that it should be allowed to perform given the Bundler's scope.

When `ICompoundV3(instance).supplyTo` is executed:

- If `asset != ICompoundV3(instance).baseToken()` it will supply collateral to Compound
- If `asset == ICompoundV3(instance).baseToken()` and `amount > ICompoundV3(instance).borrowBalanceOf(init` the caller will supply some amount (depending on the user's debt balance) instead of just repaying the debt

When `ICompoundV3(instance).withdraw` or `ICompoundV3(instance).withdrawFrom` is executed:

- If `asset == ICompoundV3(instance).baseToken()` and `amount > ICompoundV3(instance).balanceOf(address(th` (where `address(this)` in our case is the Bundler itself) it will borrow some amount of the `asset`

Recommendation: Morpho should revert the operation if the specified input parameters will lead to the execution of a Compound operation that is not in the scope of the Bundler.

- `compoundV3Repay` should only allow the caller to repay his/her debt and not supply tokens
- `compoundV3Withdraw` and `compoundV3WithdrawFrom` should only allow the caller to withdraw his/her balance and not borrow tokens

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/316

3.2.4 `AaveV3OptimizerMigrationBundler aaveV3OptimizerApproveManagerWithSig` should add support to the `skipRevert`

Severity: Low Risk

Context: AaveV3OptimizerMigrationBundler.sol#L63-L73

Description: All the functions that involve the execution of an underlying function via signature support a boolean flag called `skipRevert`.

Such flag allows the user to avoid reverting the whole multicall transaction if the signature has been already consumed (a typical scenario would be to prevent a revert because of front running).

The `AaveV3OptimizerMigrationBundler aaveV3OptimizerApproveManagerWithSig` function is the only function that is not offering the support for such behavior. Because of this, a multicall execution could be frontrunned the user/integrator would see his transaction reverted.

Recommendation: Morpho should consider adding the `skipRevert` flag to the `AaveV3OptimizerMigrationBundler aaveV3OptimizerApproveManagerWithSig`

```
function aaveV3OptimizerApproveManagerWithSig(
    bool isApproved,
    uint256 nonce,
    uint256 deadline,
    Types.Signature calldata signature,
+   bool skipRevert
) external payable {
-   AAVE_V3_OPTIMIZER.approveManagerWithSig(initiator, address(this), isApproved, nonce, deadline, signature);
+   try AAVE_V3_OPTIMIZER.approveManagerWithSig(initiator, address(this), isApproved, nonce, deadline,
↪   signature) {}
+   catch (bytes memory returnData) {
+       if (!skipRevert) _revert(returnData);
+   }
}
```

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/321

3.2.5 `AaveV2MigrationBundler.aaveV2Withdraw` and `AaveV3MigrationBundler.aaveV3Withdraw` allow the user to send their funds to `address(0)` locking them

Severity: Low Risk

Context: AaveV2MigrationBundler.sol#L45-L47, AaveV3MigrationBundler.sol#L44-L46

Description: Both the Aave v2 and Aave v3 protocols do not perform any checks on the `receiver` parameter when funds are withdrawn from the pools. This means that a user could mistakenly specify `receiver` equal to `address(0)` and lock those funds forever.

Like Morpho is already doing in other functions that are transferring tokens (see for example the various transfer functions implemented in `TransferBundler`), they should add a sanity check on the `receiver` input parameter in both the `aaveV2Withdraw` and `aaveV3Withdraw` implementations.

Recommendation: `AaveV2MigrationBundler.aaveV2Withdraw` and `AaveV3MigrationBundler.aaveV3Withdraw` should revert if `receiver == address(0)` to avoid locking the tokens withdrawn from the Aave v2 and Aave v3 pools.

Morpho

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/327

3.3 Gas Optimization

3.3.1 `CompoundV2MigrationBundler` can avoid checking the result of Compound v2 execution

Severity: Gas Optimization

Context: CompoundV2MigrationBundler.sol#L57-L59, CompoundV2MigrationBundler.sol#L71-L73

Description: The current implementation of `CompoundV2MigrationBundler` is checking the returned value of `repayBorrowBehalf` and `redeem` when `compoundV2Repay` and `compoundV2Redeem` are executed.

In reality, both `Cerc20.repayBorrowBehalf`, `Cerc20.redeem` and `CEther.redeem` will always return `NO_ERROR`.

Because of this, checking the result of the execution is meaningless and a waste of gas.

Morpho should also remove the comment found in both the function

```
// Doesn't revert in case of error.
```

Such a comment is, in fact, false because all the Compound v2 functions executed by the bundler will indeed revert in case of any error.

Recommendation: Morpho should

- Remove the false comment "// Doesn't revert in case of error.". The Compound v2 functions will revert if something wrong happens
- Remove the `require` check done on the Compound v2 execution result, given that they will always return `NO_ERROR`

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-labs/morpho-blue-bundlers/pull/312`

3.4 Informational

3.4.1 Consider forcing the `ERC4626Bundler` operations to transfer shares/underlying directly to the bundler itself

Severity: Informational

Context: ERC4626Bundler.sol#L38, ERC4626Bundler.sol#L58, ERC4626Bundler.sol#L74, ERC4626Bundler.sol#L89

Description: Each function implemented by the `ERC4626Bundler` will transfer the minted `shares` or withdrawn `underlying` to a `receiver` specified by the caller.

The role of the bundlers (at least in the Morpho Blue context) should be to eventually interact with the Morpho Blue protocol and there is no reason for the user to specify an external `receiver` that would anyway perform an additional `TransferBundler` action (later in the multicall execution) to transfer back again the shares/underlying back to the bundler to be able to perform the end action toward the Morpho Blue protocol.

Because of this reason, Morpho should consider removing the support for the arbitrary `receiver` and replace such address with the bundler's address itself.

The user would be "forced" in any way to add, at the end of the multicall operation list, a `TransferBundler` action for each asset he/she has interacted with to be sure to have left no token in the bundler.

Here's a possible re-implementation of the `ERC4626Bundler` contract following the recommendations:

```
-    function erc4626Mint(address vault, uint256 shares, address owner) external payable {
+    function erc4626Mint(address vault, uint256 shares) external payable {
-        require(owner != address(0), ErrorsLib.ZERO_ADDRESS);
-        /// Do not check `owner != address(this)` to allow the bundler to receive the vault's shares.

-        shares = Math.min(shares, IERC4626(vault).maxMint(owner));
+        shares = Math.min(shares, IERC4626(vault).maxMint(address(this)));

        address asset = IERC4626(vault).asset();
        uint256 assets = Math.min(IERC4626(vault).previewMint(shares), ERC20(asset).balanceOf(address(this)));
```

```
        require(assets != 0, ErrorsLib.ZERO_AMOUNT);

        // Approve 0 first to comply with tokens that implement the anti frontrunning approval fix.
        ERC20(asset).safeApprove(vault, 0);
        ERC20(asset).safeApprove(vault, assets);
-        IERC4626(vault).mint(shares, owner);
+        IERC4626(vault).mint(shares, address(this));
    }

-    function erc4626Deposit(address vault, uint256 assets, address owner) external payable {
+    function erc4626Deposit(address vault, uint256 assets) external payable {
-        require(owner != address(0), ErrorsLib.ZERO_ADDRESS);
-        /// Do not check `owner != address(this)` to allow the bundler to receive the vault's shares.

        address asset = IERC4626(vault).asset();

-        assets = Math.min(assets, IERC4626(vault).maxDeposit(owner));
+        assets = Math.min(assets, IERC4626(vault).maxDeposit(address(this)));
        assets = Math.min(assets, ERC20(asset).balanceOf(address(this)));

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

        // Approve 0 first to comply with tokens that implement the anti frontrunning approval fix.
        ERC20(asset).safeApprove(vault, 0);
        ERC20(asset).safeApprove(vault, assets);
-        IERC4626(vault).deposit(assets, owner);
+        IERC4626(vault).deposit(assets, address(this));
    }

-    function erc4626Withdraw(address vault, uint256 assets, address receiver) external payable {
+    function erc4626Withdraw(address vault, uint256 assets) external payable {
-        require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
-        /// Do not check `receiver != address(this)` to allow the bundler to receive the underlying asset.

        assets = Math.min(assets, IERC4626(vault).maxWithdraw(initiator));

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

-        IERC4626(vault).withdraw(assets, receiver, initiator);
+        IERC4626(vault).withdraw(assets, address(this), initiator);
    }

-    function erc4626Redeem(address vault, uint256 shares, address receiver) external payable {
+    function erc4626Redeem(address vault, uint256 shares) external payable {
-        require(receiver != address(0), ErrorsLib.ZERO_ADDRESS);
-        /// Do not check `receiver != address(this)` to allow the bundler to receive the underlying asset.

        shares = Math.min(shares, IERC4626(vault).maxRedeem(initiator));

        require(shares != 0, ErrorsLib.ZERO_SHARES);

-        IERC4626(vault).redeem(shares, receiver, initiator);
+        IERC4626(vault).redeem(shares, address(this), initiator);
    }
```

Note that depending on the ERC4626 vault implementation, `vault.maxMint` and `vault.maxDeposit` could return an unexpected result if multiple `mint` and `deposit` operations are performed on the same `vault` and `asset` during the same multicall execution without transferring the minted shares from the bundler.

Recommendation: Morpho should consider removing the support of the arbitrary `receiver` parameter from all the bundler's functions, forcing the user to send the minted shares or withdrawn underlying to the bundler itself.

Tokens left in the Bundler can be transferred at the very end of the multicall execution.

Morpho

Issue proposal is acknowledged but not implemented because there are usecases where it is desired to withdraw directly from an ERC4626 to the initiator (or whatever the `receiver`)

3.4.2 Morpho should consider adopting the same behavior across all the bundlers to be more reliable and reduce the possible edge cases that could lead to reverts or security pitfalls

Severity: Informational

Context: Multiple functions across different Bundlers

Description: Each underlying protocol (Aave v2, Aave v3, Aave (Morpho) Optimizer, Compound v2, Compound v3, ERC4626 Vaults, ...) has different requirements, implementation logics and edge case scenarios.

To reduce the revert and exploit surface and provide a common and reliable experience to the end user (EOA, Smart Wallet, Integrator, ...), Morpho should try to adopt, as much as possible, the same behavior across all the bundlers.

Discussion about `amount` input parameter

Each bundler function usually has a user's input parameter `amount`. The "meaning" of such variable depends mostly on the Bundler and the function itself.

On migration bundlers, it could represent the number of tokens to be repaid or withdrawn, on `MorphoBundler` or `ERC4626Bundler` could be the number of tokens to supply or shares to mint and so on.

Each Bundler has a non-common (across bundler) logic to bound such `amount` to a max upper bound and, moreover, each underlying protocol has a different implementation of such logic to handle edge cases.

For example: Aave allows the user to specify an `amount` greater than the balance (or the borrow amount) because the internal `withdraw` (`repay`) logic will automatically bound such amount to the user balance (or debt). Compound, Morpho Blue and ERC4626 do not adopt such behavior and "allows" the user to fail and revert the transaction if the specified amount does not strictly respect the limits.

Both behaviors are acceptable, but Morpho should adopt a common one across all the bundlers, given that they are wrapping all those protocols under the same "Bundler" umbrella.

If Morpho wants to avoid any possible revert and provide a secure and exact approval (see the following section for more details) it should spend some additional gas to properly query the underlying protocol and retrieve the exact bounds to limit the input `amount` to. If such path is chosen, one question that should be added to the mix would be which could be the side effects, down the multicall execution, to have bound the executed the operation with a value lower (because of the bounds) compared to the one that the user had specified as the input parameter.

Revert when `amount` is equal to zero

To provide a better DX (reverting with the same behavior and error across all the bundlers functions) and provide a better UX (revert as early as possible avoiding performing external calls) all the bundlers function that interact with the underlying protocol should revert early if the new calculated amount is equal to `0`.

Discussion about Token approval/allowance

Unlike for the ERC4626Bundler that is performing an exact approval

```
ERC20(asset).safeApprove(vault, 0);
ERC20(asset).safeApprove(vault, assets);
```

the other bundlers are always performing an infinite approval

```
// for Aave v2/v3, Compound v2/v3, Aave v3 Optimizer Bundlers
_approveMaxTo(...);

// for Morpho Bundler
_approveMaxMorpho(...);
```

Morpho should consider adopting only one approach across all the bundlers. If the max approval is chosen, Morpho should consider resetting the approval to zero after the execution of the function to provide a better security for the caller. If the "exact" approval is chosen, Morpho should consider resetting anyway the approval in case that the underlying protocol has "consumed" less token than the one approved. This behavior could avoid possible reverts for tokens like `USDT` (that would revert when you try to approve an amount `M > 0` when there's an existing allowance `> 0`)

Recommendation: Morpho should consider adopting a common approach and logic across all the bundlers, regarding both the upper bound of the `amount` input parameter and the bundler's allowance.

Important note: if Morpho chooses to dynamically update the `amount` specified by the user, Morpho should also warn the user that some funds that have been transferred to the Bundler could be left on the Bundler itself if not swept at the end of the multicall. Lets for example say that the user wants to repay 100 tokens of debt, but the debt of the user is just 50 tokens. The user has already transferred the 100 tokens, but the Bundler (the underlying protocol) will just "pull" 50 token (the needed amount) and the remaining delta of 50 token will be left on the Bundler. If the user doesn't sweep those tokens at the end of the multicall, the following user could use "steal" them.

Cantina:

With the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/337 Morpho has decided to adopt the "max approval" on every bundle. All the bundlers now will give `type(uint256).max` allowance to the target contract (the one that needs to pull funds).

As already mentioned, in the issue, such implementation should be safe and fine as long as:

- The `target` contract is a well-known and safe contract. Once the approval is made, the contract can pull infinite funds owned by the Bundler.
- Morpho document this behavior and warn the users
- Morpho is aware that some ERC20 tokens that do not follow the ERC20 standard could not work properly with the current implementation. See weird-erc20 for some possible examples.

3.4.3 All the migrators Bundlers should adopt the same behavior and comply with the role of the bundler itself that has to migrate funds from the origin to a Morpho Blue market

Severity: Informational

Context: AaveV2MigrationBundler.sol#L45-L47, AaveV3MigrationBundler.sol#L44-L46, AaveV3OptimizerMigrationBundler.sol#L47-L52, AaveV3OptimizerMigrationBundler.sol#L59-L61, CompoundV2MigrationBundler.sol#L66-L74, CompoundV3MigrationBundler.sol#L41-L43, CompoundV3MigrationBundler.sol#L50-L52

Description: Unlike Aave v2, Aave v3 and Aave v3 Optimizer, both Compound v2 and Compound v3 do not allow the caller to specify an arbitrary `receiver` that will receive the tokens that have been withdrawn from the pool.

While allowing the user/integrator to specify a custom `receiver` in the Aave flavor of the migrator allows Morpho to offer a better flexibility, it's also true that the role of the Migrator Bundler is to migrate funds from the origin (Aave) to Morpho Blue. Because of the role of the Bundler, it does not make sense to allow the user to specify a `receiver` that is not the Bundler itself (that later will supply those funds to the appropriate Morpho Blue market).

In addition to this reason, removing such flexibility allows Morpho to make all the migration bundlers behave the same in a consistent and predictable way.

Recommendation: Morpho should remove the `receiver` input parameters from the Aave migration Bundlers, forcing the user/integrator to withdraw those funds to the bundler itself to be later migrated to a Morpho Blue market (or manually transferred to a `receiver` via an explicit `TransferBundler` action if needed).

Here is a pseudocode example of the updated code:

```
-    /// @notice Withdraws `amount` of `asset` on AaveV2, on behalf of the initiator, transferring funds to
↪  `receiver`.
+    /// @notice Withdraws `amount` of `asset` on AaveV2, on behalf of the initiator, transferring funds to
↪  bundler.
     /// @dev Initiator must have previously transferred their aTokens to the bundler.
     /// @dev Pass `amount = type(uint256).max` to withdraw all.
-    function aaveV2Withdraw(address asset, uint256 amount, address receiver) external payable {
+    function aaveV2Withdraw(address asset, uint256 amount) external payable {
-        AAVE_V2_POOL.withdraw(asset, amount, receiver);
+        AAVE_V2_POOL.withdraw(asset, amount, address(this));
     }
```

This approach should be applied to `AaveV2MigrationBundler.aaveV2Withdraw`, `AaveV3MigrationBundler.aaveV3Withdr` `AaveV3OptimizerMigrationBundler.aaveV3OptimizerWithdraw` and `AaveV3OptimizerMigrationBundler.aaveV3Optimi`

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/327.

More natspec documentation on the bundler's behavior has been added by the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/330

3.4.4 Each Bundler function should be properly documented to warn the user/integrator about the requirements, consequences and side effects

Severity: Informational

Context: Multiple functions across all the Bundlers

Description: Each Bundler's functions have different requirements, consequences and possible side effects.

Some examples of requirements:

- User must have given allowance to the Bundler to perform "pull" of the needed amount of tokens or spend the allowance
- User must have transferred the needed amount of tokens before the execution
- ...

Some examples of consequences:

- Withdrawn/claimed funds must be transferred from the Bundler to a receiver (via a proper action inside the multicall)
- ...

Some examples of side effects:

- More funds than anticipated (specified as the `amount` input parameter) by the user could have been withdrawn. The user must "append" a transfer function at the end of the multicall execution
- Funds that are not "swept" at the end of the `multicall` can be "stolen"/used by the next multicall's user or by directly executing `TransferBundler.erc20Transfer` or `TransferBundle.nativeTransfer`
- ...

Currently, not all the requirements, consequences and possible side effects are correctly listed in all the bundler's function.

Recommendation: Morpho should iterate over **all** the functions of **all** the Bundlers and document all the requirements and possible side effects of the usage of such function.

Morpho

Fixed by https://github.com/morpho-labs/morpho-blue-bundlers/pull/330

3.4.5 Enforce the `_checkInitiated()` check in all the functions that would revert if `initiator` is equal to `address(0)`

Severity: Informational

Context: Description:

There are multiple functions across the codebase that contains the following natspec comment

```
/// @notice Warning: should only be called via the bundler's multicall function.
```

Usually, those functions require that the `initiator` (stored in `BaseBundler`) has been initialized, otherwise the underlying external call will eventually revert.

The tradeoff in gas to explicitly add such a check is neglectable compared to the total amount of gas that would be sued by the whole multicall transaction.

Recommendation: Morpho should consider enforcing such check to

- Revert early and save user/integrator gas
- Ensure that the underlying external call will not act in an unpredictable and possibly harmful way

Here's a pseudocode code example that Morpho could use:

```
    /// ... function's natspec
-    /// @notice Warning: should only be called via the bundler's `multicall` function.
    /// ... function's natspec
    function functionThatRequireInitiatorInitialized(...params) external payable {
+        _checkInitiated();

        // ... function logic
    }
```

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/313

3.4.6 `StEthBundler.stakeEth` should revert early if `amount` is equal to `0`

Severity: Informational

Context: StEthBundler.sol#L44

Description: The `StEthBundler.stakeEth` is not performing any sanity check on the value of `amount` because, as the dev comment says: "Lido will revert with ZERO_DEPOSIT in case amount == 0.".

This statement is indeed true, but such check is anyway performed by both `wrapStEth` and `unwrapStEth` even if the underlying `wsETH` function will also perform that check.

The `StEth` contract is an upgradable contract, and Morpho should not make any trust assumptions about the future behavior and checks done by the Lido contract. By adding the sanity check on the `amount` value in the `stakeEth` function will allow Morpho to

- Be coherent with the logic and implementation followed by `wrapStEth` and `unwrapStEth`
- Revert early and save user's gas if `amount == 0` (it will avoid performing an external call that will anyway revert)
- Be future-proof in case that Lido will change the logic and checks performed by an upgraded version of their implementation contract

Recommendation: Morpho should consider reverting early if `amount` is equal to `0` when `StEthBundler.stakeEth` is executed.

```
    function stakeEth(uint256 amount, address referral) external payable {
        amount = Math.min(amount, address(this).balance);

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

-        // Lido will revert with ZERO_DEPOSIT in case amount == 0.
        IStEth(ST_ETH).submit{value: amount}(referral);
    }
```

Morpho

Fixed in https://github.com/morpho-labs/morpho-blue-bundlers/pull/307

Cantina:

The issue has been addressed in the provided PR

3.4.7 `StEthBundler` should retrieve the `stETH` address directly from the `wsETH` contract

Severity: Informational

Context: StEthBundler.sol#L31-L36

Description: When the `StEthBundler` contract is deployed and initialized, the `ST_ETH` immutable variable is initialized with the constructor's input parameter `stEth`.

To avoid any possible human error, Morpho could instead retrieve such address directly by calling `IWstEth(wstEth).stETH()` that will return the `stETH` address used by the wrapper contract.

By doing this, Morpho will enforce passively the following check:

- the `ST_ETH` variable has been initialized with the correct and real address of `stETH`
- the `wstEth` is not equal to `address(0)`

Recommendation: Morpho should remove the `stEth` input parameter from the `constructor` and initialize the `ST_ETH` variable by retrieving such address directly from the `wstEth` contract.

```
-constructor(address stEth, address wstEth) {
+constructor(address wstEth) {
-    ST_ETH = stEth;
+    ST_ETH = IWstEth(wstEth).stETH();
    WST_ETH = wstEth;

    ERC20(ST_ETH).safeApprove(WST_ETH, type(uint256).max);
}
```

Cantina:

The recommendations have been implemented in the PR `https://github.com/morpho-labs/morpho-blue-bundlers/pull/324`

3.4.8 Consider adapting renaming the `MorphoBundler` functions input parameters to follow the same nomenclature used by the underlying `IMorpho` functions

Severity: Informational

Context: MorphoBundler.sol#L74-L80, MorphoBundler.sol#L95-L100, MorphoBundler.sol#L116, MorphoBundler.sol#L147, MorphoBundler.sol#L157, MorphoBundler.sol#L165-L171, MorphoBundler.sol#L178

Description: The `MorphoBundler` is a bundler contract that wrap the Morpho Blue contract functions and allows the user to call the lending protocol functions during a multicall execution.

The input parameters of the `MorphoBundler` functions do not follow the same nomenclature used by the `Morpho` (Morpho Blue) functions, and this could create confusion when the integrators or users will interact with it.

Recommendation: Morpho should consider renaming the `MorphoBundler` function's input parameters to follow the same nomenclature used by the `Morpho` (Morpho Blue) functions.

Morpho

Fixed in `https://github.com/morpho-labs/morpho-blue-bundlers/pull/314`

Cantina:

Morpho has decided to rename the `seizedCollateral` parameter in `morphoLiquidate` to `seizedAssets`. Usually assets/shares name are used to refer to the same token, while in the liquidation process the seized one is the `collateralToken` while the repaid shares refer to the `loanToken`.

Morpho

`assets` is a unite and thus can be used to any token. We'll keep this naming.

3.4.9 Consider to better documenting the usage and meaning of the `skipRevert` flag

Severity: Informational

Context: MorphoBundler.sol#L59, UrdBundler.sol#L17, PermitBundler.sol#L16, EthereumPermitBundler.sol#L18

Description: The `skipRevert` is a boolean flag that is used by different bundlers to avoid reverting the whole multicall transaction if the internal call of the bundler's function reverts and `skipRevert == true`.

The current natspec documentation found in the functions that have `skipRevert` as a function's parameter describes the flag as following (depending on the function's usage):

- `/// @dev Pass skipRevert == true to avoid reverting the whole bundle in case the signature expired.`
- `/// @dev Pass skipRevert == true to avoid reverting the whole bundle in case the proof expired.`

While during the execution of the function's logic, the `skipRevert` flag does indeed skip the revert of the whole transaction, the natspec documentation is inaccurate and should be updated with more details.

Let's make an example and look at `EthereumPermitBundler.permitDai`

```
    /// @notice Permits DAI from sender to be spent by the bundler with the given `nonce`, `expiry` & EIP-712
    /// signature's `v`, `r` & `s`.
    /// @notice Warning: should only be called via the bundler's `multicall` function.
    /// @dev Pass `skipRevert == true` to avoid reverting the whole bundle in case the signature expired.
    function permitDai(uint256 nonce, uint256 expiry, bool allowed, uint8 v, bytes32 r, bytes32 s, bool
↪   skipRevert)
        external
        payable
    {
        try IDaiPermit(MainnetLib.DAI).permit(initiator, address(this), nonce, expiry, allowed, v, r, s) {}
        catch (bytes memory returnData) {
            if (!skipRevert) _revert(returnData);
        }
    }
```

If the `IDaiPermit(MainnetLib.DAI).permit` reverts because the signature is expired (that would usually mean that the deadline has been already reached) and the whole multicall execution relies on the success of the `DAI.permit` operation, the whole transaction would anyway revert, even if `skipRevert` is equal to `true`. This would happen because at some point during the multicall execution, some operation would have needed the allowance that have been negated when `DAI.permit` have been executed.

In general, the `skipRevert` flag is used to avoid the revert of the whole transaction in edge case like front running or double-executing the permissionless function that uses a signature that will be voided after the execution.

Recommendation: Morpho should consider to properly documenting the usage of the `skipRevert` flag.

Cantina:

The recommendations have been implemented in the PR https://github.com/morpho-labs/morpho-blue-bundlers/pull/302

3.4.10 Consider renaming the `WNativeBundler` contract to `WETHBundler` given that the current implementation will only work with `WETH` tokens.

Severity: Informational

Context: WNativeBundler.sol

Description: The `WNativeBundler` contract will be used to manage the interactions with the network's wrapped native token. While the contract aims to be something that would work with the native token of **any** network, in reality is designed to only work with `WETH` or contracts that offer the same function's signature and logic.

Morpho should not take for granted that all the native tokens of the different chains will follow the same specifications implemented by `WETH`.

Recommendation: Morpho should consider the following recommendations:

- Rename the contract to `WETHBundler`
- Update the natspec documentation of the contract and all the functions natspec/comments to refer to `WETH` instead of "native token"
- Rename the `WRAPPED_NATIVE` immutable variable to `WETH_TOKEN`
- Rename the `wrapNative` function to `wrapETH`
- Rename the `unwrapNative` function to `unwrapWETH`

Cantina:

With the PR `https://github.com/morpho-labs/morpho-blue-bundlers/pull/305` Morpho has decided not to follow the recommendations but document that the term "wrapped native" refers to forks of `WETH`

3.4.11 Natspec documentation issues: missed parameters, typos or suggested updates

Severity: Informational

Context:

- IMulticall.sol
- IMorphoBundler.sol
- EthereumPermitBundler.sol#L15-L18
- AaveV2MigrationBundler.sol
- AaveV3MigrationBundler.sol
- AaveV3OptimizerMigrationBundler.sol
- CompoundV2MigrationBundler.sol
- CompoundV3MigrationBundler.sol
- ERC4626Bundler.sol
- ERC4626Bundler.sol#L24 + ERC4626Bundler.sol#L44
- MorphoBundler.sol
- MorphoBundler.sol#L71
- Permit2Bundler.sol
- PermitBundler.sol
- StEthBundler.sol
- TransferBundler.sol
- UrdBundler.sol
- WNativeBundler.sol

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

- IMulticall.sol: miss both the natspec documentation for the `interface` and the `function` declarations
- IMorphoBundler.sol: miss the natspec documentation for the `interface` declaration
- EthereumPermitBundler.sol#L15-L18: is missing the natspec documentation for all the input parameters
- EthereumPermitBundler.sol#L15-L18: should better document the `allowed` parameter. When the value is equal to `true` the `msg.sender` is giving infinite allowance to the Bundler.
- AaveV2MigrationBundler.sol: is missing the full natspec documentation for state/constants/immutable variables, the `constructor` and functions
- AaveV3MigrationBundler.sol: is missing the full natspec documentation for state/constants/immutable variables, the `constructor` and functions

- AaveV3OptimizerMigrationBundler.sol: is missing the full natspec documentation for state/constants/immutable variables, the `constructor` and functions
- CompoundV2MigrationBundler.sol: is missing the full natspec documentation for state/constants/immutable variables, the `constructor` and functions
- CompoundV3MigrationBundler.sol: is missing the full natspec documentation for state/constants/immutable variables, the `constructor` and functions
- ERC4626Bundler.sol: miss the natspec documentation for the functions input parameters
- ERC4626Bundler.sol#L24 + ERC4626Bundler.sol#L44: the `owner` parameter should be renamed `receiver` to be compliant with the `IERC4626` terminology
- MorphoBundler.sol: `onMorphoSupply`, `onMorphoSupplyCollateral`, `onMorphoRepay` and `onMorphoRepay` natspec should declare that is inherited by the Morpho Blue callback interfaces imported from IMorphoCallbacks.sol
- MorphoBundler.sol: miss the natspec documentation for the functions input parameters
- MorphoBundler.sol#L71: the `@notice` remark about the `permit2` action should be removed. The user has to manually transfer the tokens before performing the action
- Permit2Bundler.sol: miss the natspec documentation for the `permit2TransferFrom` function's parameters
- PermitBundler.sol: miss the natspec documentation for the `permit` function's parameters
- StEthBundler.sol: miss the natspec documentation for the `constructor`, `stakeEth`, `wrapStEth` and `unwrapStEth` input parameters
- TransferBundler.sol: miss the natspec documentation for the `nativeTransfer`, `erc20Transfer` and `erc20TransferFrom` input parameters
- UrdBundler.sol: miss the natspec documentation for the `urdClaim` input parameters
- WNativeBundler.sol: miss the natspec documentation for the `constructor`, `wrapNative` and `unwrapNative` input parameters

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

Fixed by `https://github.com/morpho-labs/morpho-blue-bundlers/pull/302`