OpenZeppelin | security

Morpho Blue Periphery Audit

Morpho

November 16, 2023

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-10-09 To 2023-11-02
Languages	Solidity

Total Issues	24 (16 resolved, 3 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	3 (2 resolved, 1 partially resolved)
Low Severity Issues	10 (6 resolved)
Notes & Additional Information	10 (7 resolved, 2 partially resolved)

Scope

We audited the following repositories:

- `morpho-labs/morpho-blue-bundlers` at commit 15896cd
- `morpho-org/metamorpho` at commit 62bc29a
- `morpho-org/universal-rewards-distributor` at commit 66d877b
- `morpho-labs/morpho-blue-oracles` at commit 16f372f

In scope were the following contracts:

Morpho Blue Bundlers

```
morpho-blue-bundlers/src
├── BaseBundler.sol
├── ERC4626Bundler.sol
├── MorphoBundler.sol
├── Permit2Bundler.sol
├── PermitBundler.sol
├── StEthBundler.sol
├── TransferBundler.sol
├── UrdBundler.sol
├── WNativeBundler.sol
├── ethereum
│   ├── EthereumBundler.sol
│   ├── EthereumPermitBundler.sol
│   ├── EthereumStEthBundler.sol
│   ├── interfaces
│   │   └── IDaiPermit.sol
│   ├── libraries
│   │   └── MainnetLib.sol
│   └── migration
│       └── AaveV2EthereumMigrationBundler.sol
├── interfaces
│   ├── IMorphoBundler.sol
│   ├── IMulticall.sol
│   ├── IStEth.sol
│   ├── IWNative.sol
│   └── IWstEth.sol
├── libraries
│   ├── ConstantsLib.sol
│   └── ErrorsLib.sol
└── migration
    ├── AaveV2MigrationBundler.sol
    ├── AaveV3MigrationBundler.sol
    ├── AaveV3OptimizerMigrationBundler.sol
    ├── CompoundV2MigrationBundler.sol
```

```
    ├── CompoundV3MigrationBundler.sol
    ├── MigrationBundler.sol
    └── interfaces
        ├── ICEth.sol
        ├── ICToken.sol
        ├── ICompoundV3.sol
        └── IComptroller.sol
```

Universal Rewards Distributor

```
universal-rewards-distributor/src
├── UniversalRewardsDistributor.sol
├── UrdFactory.sol
├── interfaces
│   └── IUniversalRewardsDistributor.sol
└── libraries
    ├── ErrorsLib.sol
    └── EventsLib.sol
```

Morpho Blue Metamorpho

```
morpho-blue-metamorpho/src
├── MetaMorpho.sol
├── MetaMorphoFactory.sol
├── interfaces
│   ├── IMetaMorpho.sol
│   └── IMorphoMarketParams.sol
└── libraries
     ├── ConstantsLib.sol
     ├── ErrorsLib.sol
     └── EventsLib.sol
```

Morpho Blue Oracles

```
morpho-blue-oracles/src
├── ChainlinkOracle.sol
├── interfaces
│   ├── AggregatorV3Interface.sol
│   └── IERC4626.sol
└── libraries
    ├── ChainlinkDataFeedLib.sol
    ├── ErrorsLib.sol
    └── VaultLib.sol
```

System Overview

The audited codebase consists of four different repositories:

- Morpho Blue Bundlers
- Universal Rewards Distributor
- Morpho Blue MetaMorpho
- Morpho Blue Oracles

The following sections provide an overview of each repository.

Morpho Blue Bundlers

The bundlers package allows users to perform multiple actions atomically, within a single transaction. It is designed for EOAs, since they cannot execute bundles of actions in a row like a smart contract could. There may be one main bundler per blockchain, which modularly combines multiple bundlers in one through inheritance, or multiple ones depending on which modules are necessary.

The main entrypoint of the bundler is its `multicall` function, which executes an array of payloads via `delegatecall` against the bundler contract itself. In order to keep track of who the original `msg.sender` was, its address is kept in a storage variable (`_initiator`) and will be frequently used throughout the codebase to ensure actions are carried on against the right accounts and recipients.

The following is a brief overview of every bundler available and its capabilities:

- **Morpho Bundler:** enables users to interact with Morpho Blue markets to perform actions such as supply, borrow, repay, deposit collateral, withdraw collateral, manage authorizations via signature or even execute liquidations. If any action requires a callback to the bundler, the bundler will expect the callback data to be an array of payloads to execute via `delegatecall` without modifying the `_initiator` variable.
- **ERC-4626 Bundler:** enables users to interact with any ERC-4626-compliant vault in order to deposit into and/or withdraw from it.
- **Permit Bundler:** for EIP2612-compliant tokens, this bundler allows permit-based approvals so that the approved balances can be used in subsequent calls.

- **Permit2 Bundler:** enables integrating with Uniswap's Permit2 contract in order to move funds around without having EIP2612-compliant tokens or explicit approvals beforehand.
- **StEth Bundler:** enables users to stake their ETH on Lido to receive `stEth`, and then wrap it or unwrap it in `wstEth`.
- **Wrapped Native Bundler:** similarly to the previous one, this helper bundler enables easy wrapping and unwrapping of the chain's native currency.
- **Transfer Bundler:** allows transferring ERC-20 and the native currency into and out of the bundler contract.
- **URD Bundler:** given a universal rewards distributor contract, this bundler enables claiming rewards from it in order to use those funds in Morpho or use them as the user sees fit.
- **Migration Bundlers:** there are multiple migration bundlers specifically designed to perform operations on other lending providers such as AAVE V2, V3, Compound or even the AAVE V3 Morpho optimizer. Users can bundle a set of actions such as repaying their debt, withdrawing their collateral and then deploying the same position on Morpho. Users can even leverage free Morpho flashloans to perform these actions.

Morpho Bundlers make up a powerful system that empowers users to perform a complex set of actions across the DeFi space.

Trust Assumptions

The flexibility and power of bundlers are tied to a strong trust assumption on bundle builders. Users who are not technically sophisticated will have to trust that the UI is going to offer error-free and honest bundles for them to execute.

Bundler builders are expected to not misbehave and implement best security and development practices. For instance, at no point should there be any currency balance left in a bundler contract by the time a transaction is over. If there was, anyone could go ahead and grab that balance by leveraging the `TransferBundler` contract.

It is particularly important for ERC-4626 tokens where the ratio of assets and shares might change, potentially leaving an excess of assets on the bundler balance in some scenarios.

Finally, another relevant trust assumption regarding bundle builders is that no malicious action will be bundled, for example towards a malicious contract that could re-enter the bundle contract to grab any mid-transaction balances before the final transfer in favor of the user. Such an attack is possible because even if the user specifies a final amount of ERC-20 tokens to be transferred to him, the minimum between the specified amount and the actual balance will be transferred. A malicious re-entry could leave 1 wei of every relevant token to make sure

all remaining transfers go through without reverting. See H01 to read more about the dangers of re-entering the `multicall` function within the Bundler contract.

Universal Rewards Distributor

The `UniversalRewardsDistributor` contract enables the distribution of different reward tokens to users in a gas-efficient way. Users can claim token rewards by providing a Merkle proof that is checked against the contract root. Reward distributors can be permissionlessly deployed through the `UrdFactory` contract.

Trust Assumptions

The owner of the contract can update the root and the timelock arbitrarily, potentially stealing reward tokens accumulated in the contract. The root updaters can also update the root, but are subject to the owner-set timelock during which users can decide to claim their rewards before the root gets updated. It is possible for malicious owners or root updaters to include malicious token rewards in the root, and these should thus not be blindly trusted by users. This is especially important when claiming rewards through the bundler contract.

During this audit, the owner and the root updaters were assumed to behave in the best interest of the `UniversalRewardsDistributor` users.

Privileged roles

Two roles are involved in managing a `UniversalRewardsDistributor` contract:

- The owner can arbitrarily change the root and the timelock at any time, as well as decide who the root updaters are. The owner role can be transferred and renounced.
- The root updaters can change the root but are subject to the timelock. The owner is considered a root updater.

Morpho Blue MetaMorpho

The `MetaMorpho` contract allows users to pool tokens and allocate them among several Morpho markets. Token ownership is tracked by inheriting from ERC-4626, and depositors are given vault shares in exchange for their deposits. `MetaMorpho` contracts can be deployed permissionlessly through the use of a canonical factory. Each vault is associated with a unique token address, and is managed by several privileged roles mentioned below.

Trust Assumptions

There is a high level of trust required from users when depositing into a `MetaMorpho` vault. To protect users, the owner can set a timelock, which gives depositors time to leave in case of any upcoming change which may harm their interests, such as increasing the fee or modifying the maximum amount that can be deposited to a Morpho market (referred to as a market cap). In general, users should be wary of depositing in vaults without a sufficiently large `timelock` and should monitor for suspicious pending actions.

The risk manager and the owner of a `MetaMorpho` vault can change the cap of a Morpho market. This is sensitive as Morpho markets can be created permissionlessly, opening the possibility of malicious markets being created and their cap increased in order to rug depositors.

Generally, the different actors managing a vault are assumed to behave in the best interest of `MetaMorpho` depositors. The vaults are also assumed to be deployed through the factory.

Privileged roles

There are four privileged roles involved in the management of a `MetaMorpho` vault:

- The owner of the vault has the most powerful role, as it is able to attribute the other three. As mentioned above, it has the power to change the fee but can also set the `feeRecipient` and the `rewardsDistributor` addresses.
- The risk manager can change the cap associated with Morpho markets. The owner is considered a risk manager.
- Allocators can manage the allocation of funds across the Morpho markets within the constraints set by the caps. The owner and the risk manager are considered allocators.
- The guardian can revoke some pending modifications, including changes to the timelock, the Morpho market caps or the guardian address itself. Note that if a guardian misbehaves, there is no way to kick them out and replace them.

Morpho Blue Oracles

The Morpho Team has developed an oracle implementation which will be used through the `IOracle` interface by Morpho Blue markets in order to properly price collateral and borrowed amounts. The main uses are to enable liquidations when accounts become undercollateralized, and to enforce a maximum LLTV when borrowing against a given collateral.

This implementation is flexible enough to accommodate a wide range of complex oracles from a set of primitive Chainlink price feeds. An oracle will take any base token amount and provide

its price in any quote token amount. To calculate the effective price, up to two price feeds for the base token and two more for the quote token can be provided. The rationale behind adding up to two prices per token is to allow primitive price feeds to be chained in order to create more complex price feeds. For instance, there is no Chainlink price feed to quote the price of `stEth` in USD, so by multiplying the price of `stEth` vs `Eth` by the price of `Eth` vs USD one can achieve the desired price of `stEth` in USD. This is an optional feature.

This oracle is also compatible with ERC-4626 vault shares.

In order to avoid losing precision given the disparity between token and price feed decimals, all prices are scaled by `1e36`.

Trust Assumptions

There is a trust assumption on Chainlink price feeds providing accurate prices in a timely manner. Additionally, the deployer of the oracle and most likely the Morpho Blue market creator is expected to properly configure the price feeds in order to ensure proper behaviour of the market.

Summary of Findings

We found the codebase to be very well written, with sufficient docstrings and technical documentation, along with an abundant fuzzing test suite. Once again, we are glad to see robust security patterns in place and thoughtful considerations being applied to follow best practices.

High Severity

H-01 `multicall` Function Reentrancy May Drain User Funds

The `Bundler` contracts allow the batching of several actions into one function call, reducing the friction for users when interacting with several functions and contracts. This mechanism works by exposing a `multicall` function, which iterates over its `data` argument to perform a series of `delegatecall` operations, invoking a limited set of functions on itself. When doing so, the original `msg.sender` is stored as the `_initiator` and used in subsequent function calls. This construction is important to ensure the security of users' funds by constraining which external functions and contracts are called from the bundler. This is necessary as the bundler will over time accumulate critical privileges from users such as token allowances or the ability to manage positions in other protocols (notably AAVE and Compound).

However, the current design is vulnerable to reentrancy attacks during the execution of a multicall. Since any funds in the bundler can be stolen or any permission can be exploited when the user is the `_initiator`, users have to be extremely careful regarding the content of the bundle they execute. Any interaction of the bundler with a malicious contract at any point during bundle execution could result in their funds being stolen. Additionally, because bundles are complex objects, this makes it relatively hard to check their content on common wallet interfaces. In practice, this design is brittle as it offloads a lot of the security responsibilities to bundle builders (the frontends).

Here is an example of a scenario where this could be abused:

- An attacker could craft a list of user addresses which have granted infinite approval of their balances towards the main Morpho bundler, and/or permissions to manage their positions on Aave. The attacker then decides to create an official rewards distribution towards those users as an airdrop. The reward distributor would be created through the official Morpho factory contract, adding to its perceived legitimacy. The attacker may also have spent time before the attack building trust among the Morpho community. The airdropped token may be both fake or real, since actual liquidity could be provided to a pool to make it seem valuable. However, the token is malicious and performs a series of actions upon being claimed through a multicall bundle that includes the `URDBundler`.

- Users would go to the airdrop website which lets them know about their eligibility.
- Users would click on a claim button, which would craft a bundle including a single `urdClaim` call within the `multicall` payload, and send it to the main official Morpho bundler contract.
- This will set the `_initiator` to be `msg.sender`, which is the victim's address.
- The `urdClaim` call will get executed.
- The final step on the claim call will be to transfer the malicious ERC-20 token.
- The token maliciously re-enters the main bundler contract by leveraging any of the Morpho callbacks, or even the `morphoFlashLoan` function.
- Regardless of the re-entry point, the internal `_callback` function will be called with an arbitrary array of payloads where it will check that the initiator value has already been set, which it has, and it still equals the victim address.
- This will call the internal `_multicall` function which will execute arbitrary payloads assuming the initiator is still the victim's address, without any safety checks against the current `msg.sender`.

At this point, the attacker may decide to leverage the `TransferBundler` to pull any approved balances from the user wallet into the bundler contract and then immediately out of it into their own wallet. Alternatively, migration bundlers may be used (provided permissions have been previously granted for the bundler) to wipe AAVE or Compound positions as well.

Even though this attack vector requires victims to interact with a third-party website, the exploit is made possible by the design of the bundler contract and it is very hard to detect. This kind of attack is also possible if the official website is compromised, or if any malicious code is called at any point during the execution of the multicall. Since these contracts are intended to be re-used by third parties, and not exclusively by the Morpho team, consider minimizing this risk as follows:

- Reentrancy in the `multicall` function should be prevented by checking that `_initiator` is not already set.
- All external bundler functions should revert if `msg.sender` is not either the `_initiator` or the address of the main Morpho Blue singleton.
- All Morpho Blue callbacks should be callable only by the official Morpho Blue singleton address.

Consider applying these recommendations in order to reduce the attack surface around `multicall` reentrancy .

Update: *Resolved in pull request #313 and pull request #354 at commits 7340cdb, 1d6d412 and 136553a. The Morpho team implemented a new* `protected` *modifier on all external*

functions so that they can only be called via the `multicall` function. Additionally, all Morpho Blue callbacks can only be called now by the official Morpho Blue singleton.

Medium Severity

M-01 Unreliable Fee Accrual When Modifying Fee Values

The `MetaMorpho` contract allows the owner of the contract to set a fee. This fee is taken from the yield earned by users when lending on Morpho. This fee can be changed by calling the `submitFee` function, which sets the fee immediately if there is no `timelock` or if the fee is lower than the previous one. Otherwise, the fee is set as pending to be accepted later on, once the timelock has elapsed.

When a fee is accepted after a positive timelock has elapsed, interest is accrued and the fee is charged before updating the fee value. However, when the new fee is lower than the existing one or there is no timelock in place, the fee will be changed directly without accruing interest and charging the fee. This has two main consequences:

- If there is no timelock in place, as long as interest is not accrued by some other action, when updating the fee value, it will be used for the entire period elapsed since the last accrual, ignoring the previous fee value. This goes against the user's best interest if the new fee is larger than the previous one. For instance, if the timelock is zero and the current fee is also zero, modifying the fee to a non-zero value will actually charge fees retroactively since the last accrual to all existing users, who thought fees were zero throughout their participation in the protocol.
- If there is a timelock in place but the new fee is lower, it is the fee recipient who will miss the higher fees accrued during the time period the larger fee was in place.

Both situations make fee estimation unreliable. Consider accruing interest and charging the fees any time (before) the fee is changed, regardless of the new fee or whether there is a timelock in place.

***Update:** Resolved in pull request #140 at commit 4962092. Fee accrual logic has been moved into the inner `_setFee` function to ensure that fees are always collected before updating the fee to a new value.*

M-02 Metamorpho Vaults With ERC-777 Tokens Can Be Drained in Favor of the `feeRecipient`

Each `MetaMorpho` vault, associated with a unique token, aggregates deposits from users to deposit them in a set of Morpho markets. A vault allows its owner to set a fee that gets collected by the `feeRecipient` address. This fee is subtracted from the yield earned from Morpho by the vault depositors.

However, when the underlying asset is an ERC-777, it is possible for anyone to drain the vault to the profit of the `feeRecipient`. This can happen when transferring tokens in the `deposit`, `mint`, `withdraw` or `redeem` functions, as these can be re-entered before the `lastTotalAssets` state variable used to compute the fee gets updated. This allows the `feeRecipient` to receive the same fee in shares multiple times.

This could be used by a malicious actor to grief all the depositors by diluting them to the profit of the `feeRecipient` with a vested interest in harming Morpho's image. Even worse, if the `feeRecipient` itself became compromised, the entire vault would be drained to the profit of the attacker with no upfront cost.

Consider following the check-effects-interaction pattern in the `deposit`, `mint`, `withdraw` and `redeem` functions, and/or leveraging existing solutions such as OpenZeppelin's `ReentrancyGuard` contract.

Update: *Resolved in pull request #310. The functions have been modified so that any callback to an ERC-777 happens in a valid state.*

M-03 Bundler Slippage May Cause Unexpected Loss of Funds

Several functions in the bundler contracts implement a feature that will take the minimum between the amount specified by the user and the current balance available. The intention behind this is to allow users to specify `type(uint256).max` when they cannot be sure of the final amount so that the exact final balance will be used. This functionality is present in:

- The `TransferBundler` contract at [1] [2] [3]
- The `ERC4626Bundler` contract at [1] [2] [3]
- The `StEthBundler` contract at [1] [2] [3]
- The `WNativeBundler` contact at [1] [2]
- The `AaveV2MigrationBundler` contract at [1]

- The `AaveV3MigrationBundler` contract at [1]
- The `AaveV3OptimizerMigrationBundler` contract at [1]
- The `CompoundV2MigrationBundler` contract at [1] [2] [3]
- The `CompoundV3MigrationBundler` contract at [1]

However, this functionality is analogous to an "infinite slippage" setting from the perspective of the user: if for any reason during the execution of the bundle, something unexpected happens and some tokens are lost, the transaction would still execute successfully and only transfer back any remaining balance as long as it is at least 1 wei. Furthermore, it is not necessary for a user to specify `type(uint256).max` as the `amount` for this to happen. If they specify a concrete amount and the final balance turns out to be lower, the lower value will be used instead of reverting.

Consider giving users the option to specify a custom slippage setting which will revert if a minimum amount out is not met. This feature can still be used if users input an infinite slippage value for specific situations that may require that.

Update: *Partially resolved in pull request #314 and pull request #310 at commits 146543b, f146716, 23cdcb9 and 42f3263. Slippage protection was added for the `ERC4626Bundler`, the `MorphoBundler` and the `stakeEth` function within `StEthBundler`.*

Low Severity

L-01 DoS Attacks on ERC-4626 Mints Are Possible

One of the possibilities that the ERC-4626 bundler offers is to mint a certain amount of shares within any vault that complies with the standard. This operation will be part of an arbitrarily complex list of individual operations in the bundle.

If a specific user wants to mint a fixed number of shares, it is yet unknown how many assets will be necessary to transfer into the vault. One way around this, as shown in the relevant test case, would be to call `previewMint` first in order to estimate the assets required. Users would therefore need to transfer a specific amount of assets into the bundler first, and then mint the shares. Internally, the ERC-4626 bundler will approve the vault to pull the exact amount of assets intended to be transferred.

A malicious actor could perform a denial of service attack by frontrunning one of these multicall transactions and simply transferring 1 wei of the vault underlying asset into it. This will cause

the share price to increase, very slightly, and due to rounding will cause the multicall transaction to revert because both the asset balance and the allowance are not sufficient to perform the mint. This can also happen organically if the vault balance is increased due to yield being accrued between the moment users issue their multicall transaction and the time it gets confirmed.

When dealing with operations that involve shares, some tolerance needs to be accounted for so that there is some room for the operation to go through if exact numbers do not match up. Consider adding an extra function argument `maxAssets` to reflect what is the maximum amount of assets the user is happy to spend to get those shares in case the share price fluctuates from the time the transaction is issued until it is confirmed.

This parameter should be documented so that bundler builders transfer `maxAssets` into the bundler instead of just `assets`, and then any remaining balance should be transferred back to the user before the entire bundle is done. Since bundler-building capabilities are not just restricted to the Morpho Team, it is encouraged to provide this kind of documentation so that third-party builders are also aware of this issue and can have an effective workaround.

Update: *Resolved in pull request #310 at commits 146543b and 4e7913e. The new `maxAssets` and `minShares` parameters allow for some slippage to be taken into account when minting or depositing into an ERC-4626 vault.*

L-02 Lack of Input Validation

Throughout the codebase, there are some instances where the user can withdraw some funds from Aave and is able to specify an arbitrary recipient. The following functions contain a `receiver` address which is not checked to not be zero:

- The `aaveV2Withdraw` function within the `AaveV2MigrationBundler` contract
- The `aaveV3Withdraw` function, within the `AaveV3MigrationBundler` contract

Consider preventing the `receiver` parameter from being `address(0)` in order to avoid accidentally burning user funds.

Update: *Resolved in pull request #327 at commit bfb87bc. All Aave-related withdrawals are now enforced to be received by the bundler contract, so funds will need to be transferred out later on by leveraging other bundlers.*

L-03 Chainlink Prices May Be Stale or Incorrect

The docstrings around the `getPrice` function within the `ChainlinkDataFeedLib` state that price staleness is not checked because it is assumed that Chainlink will keep its promises on this.

However, the only promise is that the `latestRoundData` will return the data from the latest round available. If for instance, a node operator decides to stop paying fees, there will be no more rounds and thus the price will eventually become stale.

Consider checking the `updatedAt` return value from the `latestRoundData` function and enforcing a staleness check making sure no more than a certain amount of time has elapsed since the latest round was published. If the price is stale, consider using a fallback oracle. Additionally, consider deciding whether having prices equal to zero is actually a desirable outcome.

Update: *Acknowledged, not resolved.*

L-04 Missing Docstrings

Throughout the codebase, there are several parts that do not have docstrings. For instance:

- Lines 33-51 in `MorphoBundler.sol`
- Lines 28-76 in `IMetaMorpho.sol`
- Lines 26-703 in `MetaMorpho.sol`

Consider thoroughly documenting the functions (and their parameters) that are part of these contract's public API. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Acknowledged, not resolved.*

L-05 Floating Pragma

Pragma directives should be fixed to clearly identify the Solidity version with which the contracts will be compiled. Throughout the codebase. there are multiple floating pragma directives:

- The file `ConstantsLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `ErrorsLib.sol` has the `solidity ^0.8.0` floating pragma directive.

- The file `EventsLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `IMetaMorpho.sol` has the `solidity >=0.5.0` floating pragma directive.
- The file `IMorphoMarketParams.sol` has the `solidity >=0.5.0` floating pragma directive.
- The file `AggregatorV3Interface.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `ChainlinkDataFeedLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `ErrorsLib.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `IERC4626.sol` has the `solidity ^0.8.0` floating pragma directive.
- The file `VaultLib.sol` has the `solidity ^0.8.0` floating pragma directive.

Consider using a fixed pragma version.

Update: *Acknowledged, not resolved.*

L-06 Files Specifying Outdated Solidity Versions

Throughout the codebase, there are several `pragma` statements that use an outdated version of Solidity. Some statements also span several minor Solidity versions, which can lead to unpredictable behavior due to differences in features, bug fixes, deprecation, and compatibility between minor versions. For instance:

- The `pragma` statement on line 2 of `IDaiPermit.sol`
- The `pragma` statement on line 2 of `IMorphoBundler.sol`
- The `pragma` statement on line 2 of `IMulticall.sol`
- The `pragma` statement on line 2 of `IStEth.sol`
- The `pragma` statement on line 2 of `IWNative.sol`
- The `pragma` statement on line 2 of `IWstEth.sol`
- The `pragma` statement on line 2 of `ICEth.sol`
- The `pragma` statement on line 2 of `ICToken.sol`
- The `pragma` statement on line 2 of `ICompoundV3.sol`
- The `pragma` statement on line 2 of `IComptroller.sol`
- The `pragma` statement on line 2 of `IMetaMorpho.sol`
- The `pragma` statement on line 2 of `IMorphoMarketParams.sol`
- The `pragma` statement on line 2 of `IUniversalRewardsDistributor.sol`

Consider pinning the Solidity version more specifically (ideally to the latest available version) throughout the codebase to ensure predictable behavior and maintain compatibility across various compilers.

Update: *Acknowledged, not resolved.*

L-07 Old Solmate Version

The `UniversalRewardsDistributor` contract uses the `SafeTransferLib` library from Solmate to handle token transfers. However, it uses 0.6.0 which is an old version. Notably, `safeTransfer` was reported to not be compatible with some curve tokens.

Consider updating to the latest available Solmate version.

Update: *Resolved in pull request #103 at commit 68056c9.*

L-08 `maxWithdraw` and `maxRedeem` Functions Can Revert Systematically

The `MetaMorpho` contract overrides two view functions from ERC-4626, `maxWithdraw` and `maxRedeem`, allowing external users to know the maximum amount of assets or shares that they can withdraw or redeem from the vault.

However, these functions revert systematically with an underflow when more than two users have deposited in one Morpho market through the `MetaMorpho` vault. The revert will happen when the maximum withdrawable amount for the whole market across all `MetaMorpho` depositors exceeds the current `remaining` variable value. This could be an issue for external contracts relying on this function to withdraw user funds.

Consider preventing this underflow by leveraging the existing `zeroFloorSub` function from Morpho Blue, since the function's goal is just to make sure there is enough liquidity in the Morpho Blue markets to satisfy the user's full withdrawal. Further, consider adding additional testing when implementing this change.

Update: *Resolved in pull request #205 at commit f306b69. The function now subtracts the minimum between `remaining` and what can be withdrawn to avoid underflows.*

L-09 Unsupported Option to Withdraw All Collateral From Compound V3

The `compoundV3Withdraw` and `compoundV3WithdrawFrom` functions can be called on the `CompoundV3MigrationBundler` to withdraw collateral or loan tokens from a Compound V3 position. According to their docstrings, the `amount` parameter can be set to `type(uint256).max` to perform a full withdrawal.

However, Compound V3 only supports `type(uint256).max` as a full withdrawal when dealing with the base `asset`. When withdrawing the collateral token, the transaction will revert on amounts larger than `uint128`, which may lead to bundles reverting unexpectedly.

Consider modifying the docstrings to more accurately match Compound's behavior, and keeping this in mind for the bundler building logic.

Update: *Resolved in pull request #316 at commit c420d76. The `compoundV3Withdraw` function was removed, and the `compoundV3WithdrawFrom` function now internally bounds the withdrawn amount depending on whether the token is the base token or the collateral.*

L-10 CompoundV3 Full Repayment Can Mistakenly Lead to Users Supplying Tokens

The `CompoundV3MigrationBundler` allows users to migrate their Compound V3 positions by repaying their debts and withdrawing their collateral through the `compoundV3Repay` and `compoundV3Withdraw` functions respectively. The `compoundV3Repay` function allows users to pass `type(uint256).max` as `amount` to indicate their intention to "repay all" their debt. This is consistent with Compound's `Comet` contract which allows for the same behavior.

However, when passing `type(uint256).max` as `amount`, the amount sent is actually the balance of the borrowed token sitting in the bundler contract. This amount can actually be greater than the borrowed amount, in which case the user would actually supply tokens instead of ending up with a neutral position (zero amounts borrowed and supplied) on Compound. This is inconsistent with what would have happened if the user had directly called the Compound `supplyTo` function with `type(uint256).max`, which would have repaid the loan and returned the difference. In practice, this could lead to unexpected results, such as users who intended to close all their positions and migrate to Morpho supplying tokens on Compound instead.

Consider ensuring that the amount to use on repayments is never larger than the currently borrowed amount in Compound.

Update: *Resolved in pull request #316. at commit d6ab91e. The amount repaid is now capped by the borrowed balance to avoid supplying.*

Notes & Additional Information

N-01 Non-explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the clarity of the code, and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long. For instance:

- Within `Permit2Bundler.sol`, line 6 has a global import.
- Within `MetaMorpho.sol`, line 10 has a global import.

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Acknowledged, not resolved.*

N-02 Duplicated Code

Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Such errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical.

When accepting a pending root after the `locktime` has elapsed, the new root is updated, the `RootSet` event is emitted and the `pendingRoot` struct values are reset. All these actions are already performed on the internal function `_setRoot`.

Rather than duplicating code, consider leveraging the existing internal function instead.

Update: *Resolved in pull request #81 at commit 102e2bb.*

N-03 Lack of Indexed Event Parameters

Throughout the codebase, several events do not have their parameters indexed. For instance:

- The `SubmitTimelock` event of `EventsLib.sol`
- The `SetTimelock` event of `EventsLib.sol`
- The `SubmitFee` event of `EventsLib.sol`
- The `SetFee` event of `EventsLib.sol`
- The `UpdateLastTotalAssets` event of `EventsLib.sol`
- The `TimelockSet` event of `EventsLib.sol`

Consider deciding whether indexing these specific event parameters makes sense to Morpho in order to improve the ability of off-chain services to search and filter for specific events.

Update: *Partially resolved in pull request #308 at commit 44c4adb. The events `SubmitFee`, `SubmitTimelock`, `UpdateLastTotalAssets` and `TimelockSet` were left without indexed parameters. The other two include a new parameter indicating the `caller` address, which has been indexed.*

N-04 Unused Import

The `Math` library is imported by the `MorphoBundler`. However, this library is not used.

Consider removing it to improve the readability of the codebase.

Update: *Resolved in pull request #352 at commit 270dcb2.*

N-05 Incorrect Documentation

Some instances were found throughout the codebase where documentation is inaccurate or incorrect:

- Documentation on the `morphoSupply` function suggests that the function uses `Permit2`, but this is not the case.
- In both `aaveV3OptimizerWithdraw` and `aaveV3OptimizerWithdrawCollateral` functions, there is a comment stating that these functions perform a repay operation, but in reality they are withdrawing funds.

Consider updating these comments to improve the clarity of the codebase.

Update: *Resolved in pull request #302 at commits 86c9464 and 86c9464.*

N-06 Unused Function Argument

In the `MorphoBundler` contract, the `morphoLiquidate` function takes an argument called `data`. If non-empty, this `data` is used as a payload for the onMorphoLiquidate callback on `msg.sender`. However, this callback is not implemented on the Morpho bundler and there is no fallback function. If there is no intention to add such a callback, the call will revert every time a non-empty `data` argument is passed.

Consider either implementing the `onMorphoLiquidate` callback if the intention is to provide this feature to users. Alternatively, consider removing the unnecessary `data` argument.

Update: *Resolved in pull request #366 at commit 9f458f0. The `onMorphoLiquidate` callback was added to the `MorphoBundler`.*

N-07 Typographical Errors

We identified the following typographical errors in the codebase:

- "a the" should be "the".
- "The address for which rewards are claimd rewards for" should be "The address for which rewards are claimed".
- "CONSTRCUTOR" should be ["CONSTRUCTOR"].

To improve the overall readability of the codebase, consider correcting the identified errors.

Update: *Resolved in pull request #101, pull request #309 and pull request #359 at commits beb4a5b, 07b4b62 and e53f5f5.*

N-08 Lack of Security Contact

Providing a security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. Most of the contracts in the codebase have one, but the `MetaMorpho` and `MetaMorphoFactory` contracts do not.

Consider adding a NatSpec comment containing a security contact on top of the contracts' definition.

Update: *Resolved in pull request #133 at commit 9e94092.*

N-09 Gas Optimizations

Throughout the codebase, some opportunities were identified to implement gas optimizations:

- The `require(<condition>, ErrorsLib.<error>)` checks across the codebase could be replaced by `if (!<condition>) revert(ErrorsLib.<error>)`. This removes the need to ABI encode the error strings, saving gas on every check.
- Constant error strings could be replaced by custom errors, but this would require upgrading the Solidity version used.
- The `sortWithdrawQueue` function within the `MetaMorpho` contract keeps the `withdrawRank` sorted. However, it is only used as a boolean flag. It could be explicitly replaced by a boolean to avoid modifying it in the function which would save gas by avoiding a non-zero warm storage change, while still deleting it when needed.
- When staking ETH on Lido, consider checking if the `amount` is zero before making the external call. It would be cheaper than reverting after the external call is made. Additionally, Lido contracts are upgradeable and thus, it is not guaranteed that this check will always be there.
- Loops could have their counter increment within an unchecked block to save gas. Some examples can be found at [1], [2], [3] and [4].

Consider implementing these changes in order to make the protocol leaner in gas consumption for the end users.

Update: *Partially resolved in pull request #282 at commit aaf3170. The Morpho team replaced the `withdrawRank` variable with a boolean to improve gas efficiency.*

N-10 Inconsistent Retrieval of `msg.sender`

The `MetaMorpho` contract uses `_msgSender()` instead of `msg.sender` everywhere, except when emitting events. For instance:

- The `setSupplyQueue` function
- The `sortWithdrawQueue` function
- The `transferRewards` function
- The `revokeTimelock` function
- The `revokeCap` function
- The `revokeGuardian` function

Consider consistently using either `msg.sender` or `_msgSender()` in order to emit accurate event information that off-chain third-party services can rely on. Alternatively, if this is intended behavior, consider adding explicit documentation about it.

Update: *Resolved in pull request #230 at commit dca4f4f.*

Conclusion

After reviewing the four repositories involved in this audit engagement, we can attest to the overall code quality and robustness, and are glad to see best security practices implemented. We found the codebase to be highly readable, well-modularized and sufficiently documented. The repositories include exhaustive fuzzing tests, which highly improves the protocol's security and speaks about the level of commitment to security from the Morpho Team.

We encourage the team to expand the test suite to include support for tokens with a number of decimals different than the default 18, to make sure all edge cases are covered. Additionally, we would like to highlight the importance of following the same good practices when developing the bundler building logic, since a lot of the security responsibilities have been offloaded to it as indicated in the introduction.