Reserve Folio Security Review

Pashov Audit Group

Conducted by: Hals, merlinboii, t0x1c, t.aksoy

June 2nd 2025 - June 11th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **reserve-protocol/reserve-index-dtf** and **reserve-protocol/trusted-fillers** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Reserve Folio & Trusted Fillers

Reserve Folio is an index basket protocol for creating and managing portfolios of ERC20 tokens, allowing precise control over asset allocations through a modular and governance-driven rebalancing system. It uses a role-based auction process to rebalance portfolios within governance-defined parameters, allowing for controlled changes in token weights and prices.

Also, the scope included Trusted Fillers, which allow Folios to settle trades asynchronously with approved external parties, enabling integration with protocols like CoW Swap while still enforcing trading rules and restrictions through EIP-1271-based validation.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- a48f777e3f1cffecb3c71cbb5a88af73eb4f7626
- 2e3eae8fd3d352240aa48a2e2b3c0168e8229e71

fixes review commit hashes:

- 79bcf277648f3a424dea46d2d8e3045e03c43c62
- 2bc2e40955d27e786e97e93aeeb4f68d96870c83

Scope

The following smart contracts were in scope of the audit:

- `TrustedFillerRegistry`
- `CowSwapFiller`
- `Constants`
- `Folio`
- `MathLib`
- `RebalancingLib`
- `Versioned`
- `StakingVault`
- `UnstakingManager`
- `FolioDeployer`
- `GovernanceDeployer`
- `IBidderCallee`
- `IFolio`
- `IFolioDAOFeeRegistry`
- `IFolioDeployer`
- `IFolioVersionRegistry`
- `IGovernanceDeployer`
- `IRoleRegistry`

7. Executive Summary

Over the course of the security review, Hals, merlinboii, t0x1c, t.aksoy engaged with Reserve to review Reserve Folio & Trusted Fillers. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	Reserve Folio & Trusted Fillers
Repository	https://github.com/reserve-protocol/reserve-index-dtf
Date	June 2nd 2025 - June 11th 2025
Protocol Type	Index Assets

Findings Count

Severity	**Amount**
Medium	1
Low	5
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Filler state manipulation causes denial-of-service across Folio operations	Medium	Resolved
[L-01]	getBid() returns incorrect results for past or future timestamps	Low	Resolved
[L-02]	toAssets() may return misleading results when a filler is in progress	Low	Acknowledged
[L-03]	Validation in isValidSignature() prevents partial fills at good prices	Low	Resolved
[L-04]	swapActive() can be manipulated to return incorrect state	Low	Acknowledged
[L-05]	Folio.bid() strict sell amount causes DoS and auction execution loss	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Filler state manipulation causes denial-of-service across Folio operations

Severity

Impact: Medium

Likelihood: Medium

Description

As the `sync()` modifier, which reverts if `swapActive()` returns `true`, is used in many crucial functions in `Folio` and also the `FolioDAOFeeRegistry`, this can be exploited to cause a DoS on protocol operations including:

1. `mint()` and `redeem()` functions, blocking users from entering or exiting positions.
2. Auction functions, preventing the protocol from executing trades.
3. Fee distribution functions, disrupt the protocol's economic model.
4. Parameter updates: `removeFromBasket()` and `FolioDAOFeeRegistry()` folio's fee setting functions.

By performing the frontruning attack to thoes operations with `{Folio.createTrustedFiller(); filler.rescueToken();}`

The attack can be executed by front-running target transactions with a sequence of:

1. Deploy a contract that calls `Folio.createTrustedFill()` followed by `filler.rescueToken()`, which manipulates internal token balances.
2. This causes `swapActive()` to return `true`, preventing the filler from being closed and causing any subsequent `sync()`

Recommendation

Possible options:

- Restrict `rescueToken()` to be `internal` and callable only through `closeFiller()`.
- Add access control to restrict the `rescueToken()`.
- Restrict `rescueToken()` to be callable only when `swapActive()` is `false`. This could ensure that fillers cannot be locked into a fake active state indefinitely and avoid protocol-wide denial-of-service scenarios.
- Add a check to disallow rescueToken from being called during block in which the contract was initialized (block.number == blockInitialized).

8.2. Low Findings

[L-01] `getBid()` returns incorrect results for past or future timestamps

The `Folio.getBid()` function is used to retrieve auction bid parameters (sellAmount, bidAmount, and price) for an ongoing auction at a specified `timestamp`. However, while the auction price is correctly calculated based on the provided timestamp, the `sellAmount` and `bidAmount` calculations rely on the current `totalSupply()`. Since `totalSupply()` can change over time (due to minting, fee inflation, etc.), **using the current `totalSupply()` when the requested timestamp is not the current block** leads to incorrect and inconsistent bid parameters, which makes it unsafe to rely on `getBid()` for off-chain simulation or for use cases that need historical or future bid estimates.

```
function getBid(
        uint256 auctionId,
        IERC20 sellToken,
        IERC20 buyToken,
        uint256 timestamp,
        uint256 maxSellAmount
    ) external view returns
      (uint256 sellAmount, uint256 bidAmount, uint256 price) {
        return
            _getBid(
                auctions[auctionId],
                sellToken,
                buyToken,
                totalSupply(), <@ should be at the snapshot supply at timestamp
                timestamp != 0 ? timestamp : block.timestamp,
                0,
                maxSellAmount,
                type(uint256).max
            );
    }
```

Recommendation:

Introduce a mechanism to snapshot or track `totalSupply` over time and use the value corresponding to the requested `timestamp` when computing `sellAmount` and `bidAmount`. Alternatively, document clearly that `getBid()` should only be used with `timestamp == block.timestamp` to avoid incorrect assumptions.

[L-02] `toAssets()` may return misleading results when a filler is in progress

The `FoliotoAssets()` function includes balances from the `activeTrustedFill` contract. If an auction swap is in progress, this may include both the buy token and/or the sell token, depending on whether the swap is partially or fully completed. As a result, the returned asset amounts for a given share amount may become inaccurate and fluctuate based on the current state of the active filler. This could mislead consuming protocols or users relying on this **view function** for precise information.

Recommendation:

Consider implementing a mechanism to make `toAssets()` callable only when there is no active filler (e.g. `require(address(activeTrustedFill) == address(0))`), or clearly document in the function comment that asset amounts may be inaccurate when a filler is active due to inclusion of unsettled filler balances.

[L-03] Validation in `isValidSignature()` prevents partial fills at good prices

The `CowSwapFiller` contract validates orders with a strict requirement that the order's sell amount must be less than or equal to the filler's current sell amount:

```
function isValidSignature
  (bytes32 orderHash, bytes calldata signature) external view returns (bytes4) {
    --- SNIPPED ---

    uint256 orderPrice = Math.mulDiv
      (order.buyAmount, D27, order.sellAmount, Math.Rounding.Floor);
@>  require(
  order.sellAmount<=sellAmount&&orderPrice>=price,
  CowSwapFiller__OrderCheckFailed
)

    --- SNIPPED ---
}
```

This creates a timing issue between the time an order is created (off-chain via the CoW Protocol API) and the time it is settled on-chain:

- When a bot submits a CoW Protocol order, it typically uses `Folio.getBid()` to fetch the current `sellAmount` and `buyAmount`.
- Between the order creation and settlement, the available `sellAmount` can change (e.g., due to other bids or redemptions).
- If the `sellAmount` decreases, the settlement will fail even if the order price is still favorable and could be partially filled.

Consider the following scenarios:

1. `AUCTION_LAUNCHER` launches an auction to rebalance the basket from 1000 ETH to 500 ETH, and buy 1,500,000 USDT (start price: `3000 USDT/ETH`).

2. A bot listens to the auction event and then submits an order through the CoW Protocol API for `500 ETH → 1,500,000 USDT` using current values from `Folio.getBid()`.

3. Before settlement, other bidders consume part of the available ETH. The filler now only has `400 ETH` available.

4. During on-chain settlement:

   - The filler contract is created via `pre-hook` with `sellAmount = 400 ETH` and `buyAmount = 1,199,600 USDT` (at the decayed auction price).
   - So the price in the filler is set to `2999 USDT/ETH`.
   - The settlement attempts to validate the original order with `order.sellAmount = 500`.
   - The signature validation fails because `order.sellAmount: 500 <= sellAmount: 400` is `false`, even though the order price (`3000`) is better than the current price (`2999`).

5. Even if the solver could partially fill the order (e.g., with `200 ETH` for `~600,000 USDT` at the requested price) at that block, this would still be a net positive execution at a better price, but it is blocked by the above validation.

Below is an example where an order could not be fully satisfied in a single settlement and was instead partially filled:
https://explorer.cow.fi/orders/0x358c3cf03715224c26f8043c94502c4649e8c59e94d43f16a90e2f8c72e3ad41a53a13a80d72a855481de5211e7654fabdfe3526685b2543/?tab=fills

Recommendation:

Given the limited `sellToken` pull via approval and that `orderPrice` already enforces the acceptable price, factoring in both `order.buyAmount` and `order.sellAmount`, it would be appropriate to rely solely on `orderPrice` for verification and remove the `order.sellAmount <= sellAmount` check.

[L-04] `swapActive()` can be manipulated to return incorrect state

`swapActive()` is called by critical flows:

1. `stateChangeActive()` --> `swapActive()`.
2. `sync()` --> `_poke()` --> `_closeTrustedFill()` --> `closeFiller()` --> `swapActive()`.

Case1: External integrating protocol calls `stateChangeActive()` and wants to see `(false, false)` before trusting the Folio state. Suppose:

- `sellAmount` is `100`.
- `sellTokenBalance` is `98` i.e. swap is still active and only partial sell has occurred.
- Ideally external protocol should see `(false, true)`.
- Attack scenario: Attacker front runs the call to `stateChangeActive()` and donates `2` sellTokens.
- Now, inside the `swapActive()` function, `if (sellTokenBalance >= sellAmount)` evaluates to `true`, and swap is reported as completed i.e. `false` is returned.
- External protocol trusts an inconsistent state & balance.

Note that a similar attack can be mounted by donating buyTokens which would result in `minimumExpectedIn > buyToken.balanceOf(address(this))` to return `false` instead of `true`.

Case2: Ideally all `sync()` modifier protected functions should revert if swaps are active because `closeFiller()` checks that:

```
require(!swapActive(), BaseTrustedFiller__SwapActive());
```

Almost all the critical Folio functions like `mint()`, `redeem()`, `distributeFees()` etc are protected by `sync()`. But due to the same attack path as above, protocol can be fooled into believing that the swap has concluded. This allows the function calls to proceed and potentially end up in an inconsistent state.

Recommendations:

Track balance internally through accounting variables instead of using `balanceOf()` inside `swapActive()`.

[L-05] `Folio.bid()` strict sell amount causes DoS and auction execution loss

The `Folio.bid()` function enforces that bids specify a strict `sellAmount` such that `minSellAmount == sellAmount == maxSellAmount`.

```
function bid(
    //...
    uint256 sellAmount,
    //...
) external nonReentrant notDeprecated sync returns (uint256 boughtAmt) {
    Auction storage auction = auctions[auctionId];

    // checks auction is ongoing and that boughtAmt is below maxBuyAmount
    (, boughtAmt, ) = _getBid(
        --- SNIPPED ---
        sellAmount,             //> minSellAmount
        sellAmount,             //> maxSellAmount
        maxBuyAmount
    );

    --- SNIPPED ---
}
```

The available sell balance is checked at execution time in `RebalanceLib::getBid()`:

```
require(
  sellAvailable>=params.minSellAmount,
  IFolio.Folio__InsufficientSellAvailable
)
```

This creates a scenario where any change to the protocol balance state between bid creation and execution can cause bids to revert, which can occur through:

1. Other bids: Each successful bid could reduce the `sellToken` balance.
2. Redemptions: These affect both the `sellToken` balance and the `sellLimitBal` calculation, as calculated below:

```
uint256 sellLimitBal = Math.mulDiv
   (sellLimit, params.totalSupply, D27, Math.Rounding.Ceil);
 uint256
    sellAvailable = params.sellBal > sellLimitBal ? params.sellBal - sellLimitBal
```

This issue can be exploited through front-running or can occur naturally through normal protocol usage, blocking bids from executing. As auction prices decay over time, the protocol could miss favorable execution opportunities.

Consider the following scenario:

Initial state

0. `totalSupply()` 1000 shares, rebalance targeting reduction from 1000 ETH to 500 ETH, and buy 1,500,000 USDT:

   - 1 BU now consist of 1 ETH 0 USDT.
   - 1 BU traget to consist of 0.5 ETH, 1500 USDT.

1. User A submits a transaction for a full bid: 500 ETH → 1,500,000 USDT at start price (most favorable price).

2. User B front-runs with a minimal bid: 0.0003333 ETH → 1 USDT.

3. The sell token balance decreases to 999.9996667 ETH.

4. The `sellAvailable` becomes 499.9996667 ETH (999.9996667 - 500 ETH).

5. When User A's transaction executes, the check `499.9996667 >= 500` fails.

6. User A's bid reverts, and the protocol loses the opportunity to execute a large bid at a favorable price.

Recommendation

Allow a range for `sellAmount` by accepting bids with different `minSellAmount` and `maxSellAmount` values via `Folio.bid()`, allow to fill as much as is available within the range at execution time.

Or, prevent griefing by either freezing `redeem()` during active auctions, or introducing a redemption fee during auctions to disincentivize such attacks.