

Usual Sync Vault

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Om parikh, Security Researcher

June 26, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Low Risk . . . 4
  - 3.1.1 `VaultRouter` is susceptible to USD0/USDS relative depeg risk due to assuming `1 USD0 = 1 USDS = 1 USD` . . . 4
- 3.2 Gas Optimization . . . 4
  - 3.2.1 VaultRouter should declare `MAX_EXCHANGE_RATE_DEVIATION_BPS` as immutable . . . 4
- 3.3 Informational . . . 5
  - 3.3.1 `VaultRouter` lacks explicit input token validation in oracle rate calculation . . . 5
  - 3.3.2 `VaultRouter` oracle rate calculation can be simplified . . . 6
  - 3.3.3 `VaultRouter` withdraw function deviates from ERC4626 return convention . . . 7
  - 3.3.4 `VaultRouter` should include tests for exploit prevention mechanisms . . . 7
  - 3.3.5 General considerations for deployment & upgrade checklist . . . 11

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

Usual is a Stablecoin DeFi protocol that redistributes control and redefines value sharing. It empowers users by aligning their interests with the platform's success.

From Jun 20th to Jun 21st the Cantina team conducted a review of usual-ethena-vault on commit hash 2b6a8294. The review focused on the following pull requests:

- PR 62
- PR 63

The team identified a total of **7** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	1	1	0
Gas Optimizations	1	1	0
Informational	5	3	2
Total	**7**	**5**	**2**

3 Findings

3.1 Low Risk

3.1.1 `VaultRouter` is susceptible to USD0/USDS relative depeg risk due to assuming `1 USD0 = 1 USDS = 1 USD`

Severity: Low Risk

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

Description:.

- Unit of `minTokensToReceive` in deposit (usd0pp ⇒ usd0 ⇒ susds): `SUSDS * (USD0 / USDS)`, while it should ideally be in `SUSDS` since tokenOut is `SUSDS` after paraswap swap.
- Unit of `minTokensToReceive` in withdraw (susds ⇒ usd0 ⇒ usd0pp): `SUSDS * (USDS / SUSDS) = USDS` while it should be in `USD0` since tokenOut is `USD0` after paraswap swap.

If `USD0 / USDS`:

- Depegs in either direction more than `max_deviation_bps + withdraw_fee` (1%) threshold.
- Stays depegged for more than cooldown period (60 seconds).

Then, vault + router system maybe exposed to some arbitrage risk.

- `oracleRate = 1.05`, `usd0/usds = 1.0`.
- Deposit 100 suds ⇒ 100 shares for 105 usd0.
- `oracleRate = 1.05`, `usd0/usds = 0.97`.
- Computed `minTokensToReceive = 100 * 1.05 = 105` ⇒ 103.5 (after adjusting for fees + deviation).
- Actual `minTokensToReceive` = 103.5 usds converted to usd0 = `103.5 * 1/0.97 = 106.7`.

(*Note: this is one of the scenarios, to cover all scenarios exhaustively, all possible combinations of `deposit/withdraw * side_of_depege * magnitude_of_depege * value_of_fee * value_of_deviation` must be considered.*)

- Some scenarios may not be profitable due to how params may be set or or direction of depeg which can max lead to reverts of user txns due to less tokens received than min computed.
- In some scenarios, risk can be controlled with changing max deviation while in others it cannot because max deviation can't be set above 1.

Recommendation: Use appropriate oracles to get market price for `USD0` and `USDS` and adjust by that in `minTokensToReceive` calculations.

Usual: Fixed in PR 69.

Cantina Managed: Fix verified.

3.2 Gas Optimization

3.2.1 VaultRouter should declare `MAX_EXCHANGE_RATE_DEVIATION_BPS` as immutable

Severity: Gas Optimization

Context: VaultRouter.sol#L82-L84

Description: The `MAX_EXCHANGE_RATE_DEVIATION_BPS` variable in VaultRouter is set once in the constructor and never modified afterward, but is declared as a regular public state variable. This is inconsistent with its intended usage as an unchangeable configuration parameter. The variable is currently declared under the comment "Mutable constants" but behaves as an immutable value since there are no functions to update it after deployment.

Recommendation: Declare `MAX_EXCHANGE_RATE_DEVIATION_BPS` as `immutable` to make the intended behavior explicit and improve gas efficiency:

```
  ///*** Mutable constants ***///
  /// @notice The maximum deviation from the oracle rate
- uint256 public MAX_EXCHANGE_RATE_DEVIATION_BPS;
+ uint256 public immutable MAX_EXCHANGE_RATE_DEVIATION_BPS;
```

This change:

- Makes the intended immutable behavior explicit in the code.
- Saves gas by storing the value in bytecode rather than contract storage.
- Improves code clarity by matching the declaration with the actual usage pattern.
- Follows Solidity best practices for values that are set once at deployment.

Usual: Fixed in PR 66.

Cantina Managed: Fix verified.

3.3 Informational

3.3.1 `VaultRouter` lacks explicit input token validation in oracle rate calculation

Severity: Informational

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

Description: The `_computeMinTokensToReceive()` function in the VaultRouter contract uses an if-else structure that assumes any token that is not USD0 should be treated as sUSDS. This implicit assumption can lead to incorrect calculations if an unexpected token is passed to the function. The current implementation:

```
if (tokenIn == USD0) {
    oracleRate = SUSDS.previewMint(1e18);
    minTokensToReceive = (amountIn * 1e18) / oracleRate;
} else {
    oracleRate = SUSDS.previewRedeem(1e18);
    minTokensToReceive = (amountIn * oracleRate) / 1e18;
}
```

The `else` clause will execute for any token that is not USD0, potentially causing incorrect oracle rate calculations for unsupported tokens.

Recommendation: Add explicit validation to ensure only supported tokens are processed:

```
  function _computeMinTokensToReceive(
      IERC20 tokenIn,
      uint256 amountIn
  )
      internal
      view
      returns (uint256 minTokensToReceive)
  {
      uint256 oracleRate;

      if (tokenIn == USD0) {
          oracleRate = SUSDS.previewMint(1e18);
          minTokensToReceive = (amountIn * 1e18) / oracleRate;
-     } else {
+     } else if (tokenIn == IERC20(address(SUSDS))) {
          oracleRate = SUSDS.previewRedeem(1e18);
          minTokensToReceive = (amountIn * oracleRate) / 1e18;
+     } else {
+         revert InvalidInputToken(address(tokenIn));
      }

      uint256 maxDeviation =
          minTokensToReceive * MAX_EXCHANGE_RATE_DEVIATION_BPS / BPS_DIVIDER;

      return minTokensToReceive - maxDeviation;
  }
```

This change improves code safety by:

- Making the supported token types explicit.
- Preventing incorrect calculations for unsupported tokens.
- Providing clear error messages when invalid tokens are used.
- Following defensive programming practices.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.3.2 `VaultRouter` oracle rate calculation can be simplified

Severity: Informational

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

Description: The `_computeMinTokensToReceive()` function in the VaultRouter contract currently calculates conversion rates through an intermediate step that can be simplified and potentially reduce rounding errors. The current implementation:

1. Retrieves an oracle rate for a fixed amount (`1e18`).
2. Manually calculates the conversion using division and multiplication operations.

This approach introduces unnecessary complexity and potential rounding errors compared to directly using the ERC4626 preview functions that are designed for these exact calculations.

Recommendation: Simplify the oracle rate calculation by using the ERC4626 preview functions directly:

```
  function _computeMinTokensToReceive(
      IERC20 tokenIn,
      uint256 amountIn
  )
      internal
      view
      returns (uint256 minTokensToReceive)
  {
-     uint256 oracleRate;
-
      if (tokenIn == USD0) {
-         oracleRate = SUSDS.previewMint(1e18);
-         minTokensToReceive = (amountIn * 1e18) / oracleRate;
+         minTokensToReceive = SUSDS.previewDeposit(amountIn);
      } else {
-         oracleRate = SUSDS.previewRedeem(1e18);
-         minTokensToReceive = (amountIn * oracleRate) / 1e18;
+         minTokensToReceive = SUSDS.previewRedeem(amountIn);
      }

      uint256 maxDeviation =
          minTokensToReceive * MAX_EXCHANGE_RATE_DEVIATION_BPS / BPS_DIVIDER;

      return minTokensToReceive - maxDeviation;
  }
```

This change:

- Eliminates intermediate rate calculations and manual conversions.
- Reduces potential rounding errors by using the vault's native preview functions.
- Simplifies the code logic and improves readability.
- Leverages the ERC4626 standard's designed functionality for these exact use cases.

Usual: Fixed in PR 64.

Cantina Managed: Fix verified.

3.3.3 `VaultRouter` withdraw function deviates from ERC4626 return convention

Severity: Informational

Context: VaultRouter.sol#L259-L269

Description: The `withdraw()` function in VaultRouter returns the amount of USD0PP tokens received (`amountUSD0pp`) rather than the number of vault shares burned, which deviates from the standard ERC4626 convention where withdraw functions typically return the shares amount. In the ERC4626 standard, a withdraw function usually returns the number of shares that were redeemed to provide the requested assets. However, the VaultRouter's withdraw function returns the final USD0PP amount that the user receives after the withdrawal and swap process. This design choice makes sense from a user experience perspective, as users interacting with the router are primarily concerned with the final token amount they receive rather than the intermediate vault share operations.

Recommendation: Consider adding documentation to clarify this design choice:

```
  /// @inheritdoc IVaultRouter
+ /// @dev Returns the amount of USD0PP received rather than shares burned,
+ /// as users interact with final token amounts rather than vault shares
  function withdraw(
      IParaSwapAugustus augustus,
      uint256 assets,
      address receiver,
      uint256 minUSD0PPToReceiveUser,
      bytes calldata swapData
  )
      public
      whenNotPaused
      nonReentrant
      returns (uint256 amountUSD0pp)
```

This documentation helps developers understand the intentional deviation from the standard ERC4626 pattern and clarifies that the return value represents the user's final token receipt rather than intermediate vault operations.

Usual: Fixed in PR 65.

Cantina Managed: Fix verified.

3.3.4 `VaultRouter` should include tests for exploit prevention mechanisms

Severity: Informational

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

Description: The `VaultRouter` contract has implemented protections against a specific exploit that occurred on May 27, 2025, involving dust swap attacks through manipulated Uniswap pools. While the fix has been implemented through oracle-based minimum amounts and fee mechanisms, the current test suite should include fuzz tests that validate these protections under various scenarios.

The exploit involved:

1. Flash loans of large USD0++ amounts.
2. Creating pools with minimal sUSDS liquidity.
3. Converting USD0++ to USD0 at 1:1 ratio, then swapping all USD0 for minimal sUSDS.
4. Extracting USD0 from the pool and converting back to USD0++ at market rates for profit.

The implemented fix enforces oracle-based minimum amounts using `sUSDS.previewMint()` with a maximum deviation threshold and withdrawal fees to prevent profitable exploitation.

Recommendation: Include a test suite that validates the exploit prevention mechanisms:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import { BaseIntegrationTest } from "./BaseUnit.integration.t.sol";
import { MockAugustus } from "./mocks/MockAugustus.sol";
import { IParaSwapAugustus } from "../src/interfaces/IParaSwapAugustus.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

```solidity
import { IERC4626 } from "@openzeppelin/contracts/interfaces/IERC4626.sol";
import { Math } from "@openzeppelin/contracts/utils/math/Math.sol";
import { IUSD0ppMinter } from "../src/interfaces/IUSD0ppMinter.sol";
import { VaultRouter } from "../src/VaultRouter.sol";
import { StdStorage, stdStorage } from "forge-std/StdStorage.sol";
import { console } from "forge-std/console.sol";

import { ADDRESS_AUGUSTUS_REGISTRY, BPS_DIVIDER } from "../src/constants.sol";

/**
 * @title VulnerabilityDemoTest
 * @notice Tests demonstrating that the May 27, 2025 exploit vulnerability has been fixed
 * @dev This test suite validates the hotfix implementation that prevents dust swap attacks
 */
contract VulnerabilityDemoTest is BaseIntegrationTest {
    using Math for uint256;
    using stdStorage for StdStorage;

    MockAugustus public mockAugustus;
    IERC20 public USDS;

    function setUp() public override {
        super.setUp();

        USDS = IERC20(IERC4626(address(SUSDS)).asset());

        // Deploy and set up mock Augustus
        mockAugustus = new MockAugustus();

        // Mock Augustus registry validation
        vm.mockCall(
            ADDRESS_AUGUSTUS_REGISTRY,
            abi.encodeWithSignature("isValidAugustus(address)", address(mockAugustus)),
            abi.encode(true)
        );

        // Set unlimited unwrap cap for testing
        vm.startPrank(unwrapCapAllocator);
        IUSD0ppMinter(address(USD0PP)).setUnwrapCap(
            address(router), type(uint256).max
        );
        vm.stopPrank();

        // Label contracts for debugging
        vm.label(address(USD0), "USD0");
        vm.label(address(USD0PP), "USD0PP");
        vm.label(address(SUSDS), "sUSDS");
        vm.label(address(mockAugustus), "mockAugustus");
        vm.label(address(router), "router");
        vm.label(address(vault), "vault");
    }

    function setMaxExchangeRateDeviationBps(uint256 bps) public {
        stdstore.target(address(router)).sig(
            bytes4(keccak256("MAX_EXCHANGE_RATE_DEVIATION_BPS()"))
        ).checked_write(bps);
        assertEq(router.MAX_EXCHANGE_RATE_DEVIATION_BPS(), bps);
    }

    function setWithdrawFeeRateBps(uint256 bps) public {
        stdstore.target(address(vault)).sig(
            bytes4(keccak256("withdrawFeeRateBps()"))
        ).checked_write(bps);
        assertEq(vault.withdrawFeeRateBps(), bps);
    }

    // Maximum expected USD0++ price deviation
    uint256 MIN_USD0_TO_USD0PP_PRICE = 0.9e18;

    // Test the specific exploit scenario with protections enabled
    function testExploitWithProtectionsEnabled() public {
        _testExploitWithProtectionsEnabled(
            1_900_000e18, // Flash loan amount from actual incident
            0.995e18,     // Max output deviation ratio (0.5% deviation)
            0.97e18,      // USD0++ price
            0.005e18      // Fee override (0.5% fee)
```

```
    );
}

// Fuzz test to validate protections across parameter ranges
function testExploitWithProtectionsEnabledFuzz(
    uint256 L, // Flash loan amount
    uint256 r, // Max output deviation ratio
    uint256 p  // USD0++ price
) public {
    _testExploitWithProtectionsEnabled(L, r, p, 0);
}

// Core test implementation that validates:
// - Oracle-based minimum amount enforcement
// - Proper fee calculation to prevent profitable exploitation
// - Price deviation bounds
function _testExploitWithProtectionsEnabled(
    uint256 L, uint256 r, uint256 p, uint256 feeOverride
) public {
    vm.startPrank(alice);

    // Bound parameters to realistic ranges
    p = bound(p, MIN_USD0_TO_USD0PP_PRICE, 1e18); // 0.9%  p  100%
    L = bound(L, 1e18, 1e25); // 1e18  L  1e24
    r = bound(r, 0.9e18, 0.999e18); // 0.9%  maxDeviation  99.9%

    // Calculate minimum fee required to prevent profitable exploitation
    uint256 f = (1e18 * ((1e18 - p) * (1e18 - r)) + p * r - 1) / (p * r);
    if (feeOverride > 0) {
        f = feeOverride;
    }

    console.log("L: %18e", L);
    console.log("r: %18e", r);
    console.log("p: %18e", p);
    console.log("f: %18e", f);

    uint256 skL;
    uint256 u;
    {
        // Set withdrawal fees and maximum output deviation
        uint256 minWithdrawFeeRateBps = (f * BPS_DIVIDER + 1e18 - 1) / 1e18;
        uint256 maxOutputDeviationBps = (1e18 - r) * BPS_DIVIDER / 1e18;

        setWithdrawFeeRateBps(minWithdrawFeeRateBps);
        setMaxExchangeRateDeviationBps(maxOutputDeviationBps);

        // Calculate minimum sUSDS required based on oracle rate
        uint256 t = (L * 1e18 / IERC4626(address(SUSDS)).previewMint(1e18));
        skL = t - t * maxOutputDeviationBps / BPS_DIVIDER;
        u = IERC4626(address(SUSDS)).previewMint(skL);
    }

    // Setup tokens and approvals
    deal(address(USDS), alice, u);
    deal(address(USD0PP), alice, L);

    IERC20(USDS).approve(address(SUSDS), u);
    IERC20(USD0PP).approve(address(router), L);
    IERC20(vault).approve(address(router), type(uint256).max);

    // Prepare sUSDS for mock Augustus
    IERC4626(address(SUSDS)).mint(skL, address(alice));
    SUSDS.transfer(address(mockAugustus), skL);

    // Mock the manipulated pool swap result
    mockAugustus.setSwapResult(
        true,
        address(USD0),
        L, // Input: L USD0
        address(SUSDS),
        skL, // Output: skL sUSDS
        address(router)
    );

    // Execute deposit with dust swap attack attempt
```

```
        router.deposit(
            IParaSwapAugustus(address(mockAugustus)),
            USD0PP,
            L,
            1, // minSharesReceived
            0, // minAmountReceived
            address(alice),
            ""
        );

        // Transfer extracted USD0 to attacker using MockAugustus transfer function
        mockAugustus.transfer(address(USD0), alice, L);

        skip(61); // Skip cooldown period

        // Execute withdrawal
        uint256 sUSDSWithdrawAmt = vault.maxWithdraw(alice);
        uint256 USDSWithdrawAmt = IERC4626(address(SUSDS)).previewRedeem(sUSDSWithdrawAmt);

        // Mock fair market rate withdrawal swap
        deal(address(USD0), address(mockAugustus), USDSWithdrawAmt);
        mockAugustus.setSwapResult(
            true,
            address(SUSDS),
            sUSDSWithdrawAmt,
            address(USD0),
            USDSWithdrawAmt,
            address(router)
        );

        router.withdraw(
            IParaSwapAugustus(address(mockAugustus)),
            sUSDSWithdrawAmt,
            alice,
            0,
            ""
        );

        // Calculate profit and verify exploit is not profitable
        int256 profit = int256(USD0.balanceOf(alice))
            + int256(USD0PP.balanceOf(alice) * p / 1e18) - int256(u)
            - int256(L * p / 1e18);

        assertLt(profit, 0, "Attacker made a profit - exploit protection failed");
    }
}

// MockAugustus with transfer capability for testing
contract MockAugustus {
    function transfer(address token, address to, uint256 amount) external {
        IERC20(token).transfer(to, amount);
    }

    function setSwapResult(/* swap parameters */) external {
        // Mock swap implementation
    }
}
```

The test suite should validate three critical assumptions:

1. USD0PP Price Bounds: Tests should cover the expected maximum price deviation range for USD0PP (e.g., 0.9 to 1.0 ratio).
2. Maximum Output Deviation: Validate that the `MAX_EXCHANGE_RATE_DEVIATION_BPS` setting (currently 0.5%) provides adequate protection across realistic market conditions.
3. Fee Calculation: Ensure withdrawal fees are calculated correctly using the formula `(1 - p) * (1 - r) / (p * r)` to cover maximum price and output deviation scenarios.

These tests provide confidence that the implemented protections effectively prevent the exploit under various market conditions and attack parameters, while ensuring the system remains functional for legitimate users.

Usual: Fixed in PR 68.

Cantina Managed: Fix verified.

3.3.5 General considerations for deployment & upgrade checklist

Severity: Informational

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

Description:

- `console.logBytes` can be used instead of `bytesToHexString` and casting to `string memory` to avoid introducing additional dependency for critical function.
- Use certain script flags/opts when running such as `--slow`, custom gas price or higher multiplier, `--force` when running first/non-failed attempt, etc. This could also be added in `foundry.toml` in a separate deployment profile.
- Make sure to verify contracts on etherscan before executing upgrade or setting router.
- Since vault was paused for long time, and deposit/withdraw actions don't auto-harvest, an explict harvest must be batched with unpause (after `initializeV1`).
- Double check if `harvestFeeRateBps` needs to be changed before unpausing as new invariant `$.harvestFeeRateBps <= $.withdrawFeeRateBps` was introduced in PR 62.

Usual: Acknowledged.

Cantina Managed: Acknowledged.