CANTINA

Usual USD0++ Investment Vault

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Om Parikh, Security Researcher

April 9, 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 Medium Risk . . . 4
3.1.1 Incorrect fee calculation in withdraw and redeem functions . . . 4
3.1.2 Incorrect asset calculation in `maxWithdraw()` function . . . 6
3.1.3 Incorrect fee share calculation in harvest function . . . 7
3.2 Low Risk . . . 8
3.2.1 Withdraw function lacks max shares slippage parameter . . . 8
3.2.2 Cap limit in `unwrapWithCap()` not replenished by mint . . . 9
3.2.3 Retroactive fee rate application in harvest function . . . 10
3.2.4 Higher precision needed for vault fee calculation . . . 10
3.3 Gas Optimization . . . 11
3.3.1 Overly complex permit handling logic . . . 11
3.3.2 Redundant slippage checks for trusted operations . . . 12
3.4 Informational . . . 14
3.4.1 Asset address stored redundantly in vault contract . . . 14
3.4.2 Initial fee rate timestamp not set during initialization . . . 14
3.4.3 Deposit function lacks receiver parameter unlike withdraw . . . 15
3.4.4 Incorrect event parameters in deposit function . . . 16
3.4.5 Code quality improvements . . . 17

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 Mar 15th to Mar 18th the Cantina team conducted a review of usual-investment-vault on commit hash 78c96f4a. The team identified a total of **14** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	3	0
Low Risk	4	1	3
Gas Optimizations	2	2	0
Informational	5	5	0
Total	**14**	**11**	**3**

3 Findings

3.1 Medium Risk

3.1.1 Incorrect fee calculation in withdraw and redeem functions

Severity: Medium Risk

Context: WrappedDollarVault.sol#L332-L367

Summary: The `WrappedDollarVault` contract incorrectly calculates fees during withdrawal operations. The implementation of `previewWithdraw()` and `previewRedeem()` uses mathematical formulas that don't accurately represent the intended fee calculations, leading to incorrect asset/share amounts being returned to users and incorrect fee amounts.

Description: The calculations in `previewRedeem()` and `previewWithdraw()` use different mathematical approaches that result in inconsistent fee application:

1. In `previewRedeem()`, the contract:
   - Calculates fee shares directly: `feeShares = _feeOnRaw(shares, feeRateBps)`.
   - Subtracts the fee from shares: `netShares = shares - feeShares`.
   - Returns assets corresponding to remaining shares: `return super.previewRedeem(netShares)`.
2. In `previewWithdraw()`, the contract:
   - Calculates shares needed without fee: `sharesWithoutFee = super.previewWithdraw(assets)`.
   - Calculates fee on those shares: `feeShares = _feeOnRaw(sharesWithoutFee, feeRateBps)`.
   - Returns total shares needed: `return sharesWithoutFee + feeShares`.

The current implementation of `previewWithdraw()` uses the formula

```
shares = toShares(assets) * (1 + f)
```

However, the mathematically equivalent formula to

```
previewRedeem()` should be: `shares = toShares(assets / (1 - f))
```

Impact Explanation: The impact is medium. Users may receive more or fewer assets than expected when withdrawing from the vault. This discrepancy becomes more pronounced as the fee rate increases, potentially affecting users' financial positions and a loss in revenue.

Likelihood Explanation: The likelihood of this issue occurring is high as it's an inherent part of the contract's core functionality. Every withdrawal or redemption operation will be affected by this calculation error.

Recommendation: Replace the current fee calculation logic with a mathematically correct implementation.

1. Corrected fee calculation helper functions:

```
function _feeOnRaw(
    uint256 amount,
    uint256 feeBasisPoints
)
    private
    pure
    returns (uint256)
{
    return amount.mulDiv(feeBasisPoints, BPS_DIVIDER, Math.Rounding.Ceil);
}

function _feeOnTotal(
    uint256 amount,
    uint256 feeBasisPoints
)
    private
    pure
    returns (uint256)
{
    return amount.mulDiv(
        feeBasisPoints, BPS_DIVIDER - feeBasisPoints, Math.Rounding.Ceil
    );
}
```

2. Corrected `previewWithdraw()` function:

```
function previewWithdraw(uint256 assets)
    public
    view
    virtual
    override(ERC4626Upgradeable, IERC4626)
    returns (uint256)
{
    // Calculate fee amount
    WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
    uint256 fee = _feeOnTotal(assets, $.feeRateBps);

    // Calculate shares needed for assets + fee
    return super.previewWithdraw(assets + fee);
}
```

3. Corrected `previewRedeem()` function:

```
function previewRedeem(uint256 shares)
    public
    view
    virtual
    override(ERC4626Upgradeable, IERC4626)
    returns (uint256)
{
    // Convert shares to assets
    uint256 assets = super.previewRedeem(shares);

    // Calculate and deduct fee
    WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
    uint256 fee = _feeOnRaw(assets, $.feeRateBps);

    return assets - fee;
}
```

4. Consolidate `_withdrawAssets()` and `_redeemShares()` into a single `_withdraw()` function:

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
)
    internal
    virtual
    override
{
    WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
    address recipient = $.treasury;

    // Calculate fee shares
    uint256 feeShares = _feeOnRaw(shares, $.feeRateBps);

    // Call parent implementation to handle the withdrawal
    super._withdraw(caller, receiver, owner, assets, shares);

    // Mint fee shares to treasury
    if (feeShares > 0 && recipient != address(0)) {
        _mint(recipient, feeShares);
    }
}
```

By implementing these changes, the contract will correctly calculate fees and ensure users receive the expected amount of assets when withdrawing from the vault.

Usual: Fixed in PR 49.

Cantina Managed: Fix verified.

3.1.2 Incorrect asset calculation in `maxWithdraw()` function

Severity: Medium Risk

Context: WrappedDollarVault.sol#L312-L316

Summary: The `maxWithdraw()` function in the WrappedDollarVault contract incorrectly calculates the maximum amount of assets that can be withdrawn by a user. This inconsistency creates a discrepancy between the actual withdrawable assets and what the function reports, potentially causing transaction failures or unexpected behavior.

Description: There are two issues with the `maxWithdraw()` implementation:

1. The `withdraw()` function calls `super.maxWithdraw(owner)` (which refers to ERC4626Upgradeable's implementation) instead of calling the overridden `maxWithdraw()` from WrappedDollarVault. This means the fee calculation in WrappedDollarVault's `maxWithdraw()` is bypassed.

2. The fee calculation in `maxWithdraw()` doesn't align with the fee calculation in `previewWithdraw()`, which leads to inconsistent behavior.

Impact Explanation: The impact is medium. The incorrect calculation in `maxWithdraw()` can lead to a poor user experience where withdrawal transactions fail unexpectedly. It could also potentially lead to users being unable to withdraw their full entitled assets if the function underestimates the maximum amount. This creates confusion and may reduce trust in the protocol.

Likelihood Explanation: The likelihood is medium. This issue affects every user interaction with the `maxWithdraw()` or `withdraw()` function. Any user interface or integration that relies on this function to determine withdrawal limits will consistently provide incorrect information to users. This will result in either transaction failures (if the function overestimates) or suboptimal withdrawals (if it underestimates).

Recommendation:

1. Modify the `withdraw()` function to use the contract's own `maxWithdraw()` implementation, or simply make use of `super.withdraw()`:

```
    function withdraw(
        uint256 assets,
        address receiver,
        address owner
    )
        public
        override(ERC4626Upgradeable, IERC4626)
        whenNotPaused
        nonReentrant
        checkRouter
        returns (uint256)
    {
        if (assets == 0) revert ZeroAmount();
        if (receiver == address(0)) revert NullAddress();

-       uint256 maxAssets = super.maxWithdraw(owner);
-       if (assets > maxAssets) {
-           revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
-       }
-
-       uint256 shares = previewWithdraw(assets);
-       _withdrawAssets(msg.sender, receiver, owner, assets, shares);
-
-       return shares;
+       return super.withdraw(assets, receiver, owner);
    }
```

2. Reimplement `maxWithdraw()` to properly align with the fee calculation in `previewWithdraw()`:

```
function maxWithdraw(address owner)
    public
    view
    override(ERC4626Upgradeable, IERC4626)
    returns (uint256)
{
    if (paused()) return 0;

    // Get the maximum assets the user could withdraw without considering fees
    uint256 assets = super.maxWithdraw(owner);
    if (assets == 0) return 0;

    // Apply fee calculation consistent with previewWithdraw
    WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
    uint256 fee = _feeOnRaw(assets, $.feeRateBps);

    // Return the maximum assets after deducting fees
    return assets - fee;
}
```

Usual: Fixed in PR 49.

Cantina Managed: Fix verified.

3.1.3 Incorrect fee share calculation in harvest function

Severity: Medium Risk

Context: WrappedDollarVault.sol#L528-L531

Summary: The `harvest()` function in the WrappedDollarVault contract incorrectly calculates the amount of fee shares to mint. The current implementation applies the fee rate to the existing total supply, which doesn't account for the dilution created by the newly minted shares.

Description: In the current implementation of the `harvest()` function, fee shares are calculated as a percentage of the existing token supply:

```
uint256 currentSupply = totalSupply();
sharesMinted = currentSupply.mulDiv($.feeRateBps, BPS_DIVIDER, Math.Rounding.Floor);
```

This approach doesn't properly calculate the correct proportion of fee shares to mint. When new shares are minted, the total supply increases, but the fee calculation doesn't account for these new shares in the total.

For a fee rate of `f`, the correct proportion of new shares relative to the post-mint total supply should be `f/(1-f)` of the total. The current implementation simply mints `f` of the pre-mint total, which results in less than `f` of the post-mint total being fee shares. This means the protocol is earning fewer fees than intended according to the fee rate.

Impact Explanation: The impact is medium. The treasury receives fewer fee shares than it should based on the intended fee rate. This effectively reduces the protocol's fee income and benefits existing token holders at the expense of the treasury. The magnitude of this issue increases with higher fee rates.

Likelihood Explanation: The likelihood is high as this issue occurs every time the `harvest()` function is called.

Recommendation: Modify the `harvest()` function to correctly calculate the fee shares relative to the post-mint total:

```
  function harvest()
      external
      whenNotPaused
      nonReentrant
      returns (uint256 sharesMinted)
  {
      WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();

       if (!$.registryAccess.hasRole(VAULT_HARVESTER_ROLE, _msgSender())) {
          revert NotAuthorized();
      }

      if (block.timestamp < $.lastHarvestTimestamp + ONE_DAY) {
          revert HarvestTooFrequent();
      }

      uint256 currentSupply = totalSupply();
-     sharesMinted =
-         currentSupply.mulDiv($.feeRateBps, BPS_DIVIDER, Math.Rounding.Floor);
+     // Calculate fee shares using formula fS/(1-f) to get the correct proportion
+     sharesMinted = currentSupply.mulDiv(
+         $.feeRateBps,
+         BPS_DIVIDER - $.feeRateBps,
+         Math.Rounding.Ceil
+     );

      if (sharesMinted == 0) revert ZeroAmount();

      $.lastHarvestTimestamp = block.timestamp;

      _mint($.treasury, sharesMinted);

      emit Harvested(_msgSender(), sharesMinted);

      return sharesMinted;
  }
```

Usual: Fixed in PR 50.

Cantina Managed: Fix verified.

3.2 Low Risk

3.2.1 Withdraw function lacks max shares slippage parameter

Severity: Low Risk

Context: VaultRouter.sol#L223

Description: In the `VaultRouter` contract, the `deposit()` function includes a `minSharesReceived` parameter that allows users to specify the minimum number of shares they expect to receive, protecting against slippage. However, the `withdraw()` function does not offer a similar protection parameter to limit the maximum number of shares that will be burned during withdrawal.

Without a `maxSharesRedeemed` parameter in the `withdraw()` function, users have no way to limit the number of shares that might be redeemed during a withdrawal, which could lead to unexpected share costs in volatile market conditions.

Recommendation: Add a `maxSharesRedeemed` parameter to the `withdraw()` function and implement a check to ensure that the actual number of shares redeemed does not exceed this limit:

```
  function withdraw(
      IParaSwapAugustus augustus,
      uint256 assets,
      uint256 minUSD0ppToReceive,
+     uint256 maxSharesToRedeem,
      address receiver,
      bytes calldata swapData
  )
      public
      whenNotPaused
      nonReentrant
      returns (uint256 amountUSD0pp)
  {
      if (receiver == address(0)) {
          revert NullAddress();
      }

      uint256 initialTokenOutBalance =
          IERC20(VAULT.asset()).balanceOf(address(this));

-     VAULT.withdraw(assets, address(this), _msgSender());
+     uint256 sharesRedeemed = VAULT.withdraw(assets, address(this), _msgSender());
+     if (sharesRedeemed > maxSharesToRedeem) {
+         revert ExcessiveSharesRequired();
+     }

      // ... rest of function ...
  }
```

Add the corresponding error to the contract:

```
error ExcessiveSharesRequired();
```

Also update the `withdrawWithPermit()` function to pass this new parameter.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.2.2 Cap limit in `unwrapWithCap()` not replenished by mint

Severity: Low Risk

Context: VaultRouter.sol#L345

Description: In the Usd0PP contract, the `unwrapWithCap()` function allows users with the `USD0PP_CAPPED_UNWRAP_ROLE` to unwrap USD0PP tokens up to their assigned cap limit. However, there is no mechanism to replenish this cap when new tokens are minted. When a contract calls `unwrapWithCap()`, their remaining cap is decreased:

```
$.unwrapCaps[msg.sender] -= amount;
```

However, when the same contract calls `mint()` to obtain new USD0PP tokens, their unwrap cap is not increased. This could lead to a situation where a contract's unwrap capacity is exhausted even though they have a balance of USD0PP tokens, effectively causing a denial of service for unwrapping further tokens.

Recommendation: If this behavior is intentional, document and monitor the remaining unwrap caps and increase these when necessary. If not, consider implementing a mechanism to replenish the unwrap cap when users mint new USD0PP tokens. This could be done by adding code to the `mint()` function that increases the user's unwrap cap proportionally to the amount minted:

```
 function mint(uint256 amountUsd0) public nonReentrant whenNotPaused {
     Usd0PPStorageV0 storage $ = _usd0ppStorageV0();

     // revert if the bond period isn't started
     if (block.timestamp < $.bondStart) {
         revert BondNotStarted();
     }
     // revert if the bond period is finished
     if (block.timestamp >= $.bondStart + BOND_DURATION_FOUR_YEAR) {
         revert BondFinished();
     }

     // get the collateral token for the bond
     $.usd0.safeTransferFrom(msg.sender, address(this), amountUsd0);

     // update the daily USD0++ inflows
     _updateDailyUSD0pplFlow(amountUsd0, true);

+     // Increase unwrap cap if user has role
+     if ($.registryAccess.hasRole(USD0PP_CAPPED_UNWRAP_ROLE, msg.sender)) {
+         $.unwrapCaps[msg.sender] += amountUsd0;
+     }

     // mint the bond for the sender
     _mint(msg.sender, amountUsd0);
 }
```

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.2.3 Retroactive fee rate application in harvest function

Severity: Low Risk

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

Description: The current implementation of fee harvesting in the `WrappedDollarVault` contract applies fee rate changes retroactively. When the fee rate is changed between harvest operations using the `setFeeRateBps()` function, the new rate is applied to the entire period since the last harvest event. In the `harvest()` function, fees are calculated using:

```
sharesMinted = currentSupply.mulDiv($.feeRateBps, BPS_DIVIDER, Math.Rounding.Floor);
```

This calculation simply uses the current fee rate (`$.feeRateBps`) without accounting for historical rate changes. As a result, if the fee rate changes from 10 BPS to 20 BPS between harvest calls, all accumulated fees will be calculated using 20 BPS - even for the period when users expected the 10 BPS rate.

This creates an inequity for users who interacted with the vault under the expectation of the lower fee rate, as they are effectively charged at the higher rate retroactively.

Recommendation:

- Ensure harvest is called each time fees are changed so that correct fee rate is charged & add test scenarios.
- Explore continuous fee accrual or time-weighted accumulated accounting, however this might require structural changes.

Usual: Acknowledged.

Cantina Managed: Acknowledged.

3.2.4 Higher precision needed for vault fee calculation

Severity: Low Risk

Context: constants.sol#L10-L13

Description: The WrappedDollarVault contract currently uses a basis point (BPS) system for fee calculation that may lack sufficient precision for daily fee accrual. With the current setup, the fee rate is expressed in basis points (1/10_000), which might not provide adequate granularity for very low daily fees. Given that the `harvest()` function can be called daily and the expected annual percentage rate (APR) is likely only a few percent, the daily fee would be a very small fraction that could be subject to rounding errors or might not be representable within the current precision constraints. For example, a 5% APR would equate to approximately 0.0137% daily (5.00% ÷ 365), which is 1.37 basis points - at the lower end of the current precision scale.

Recommendation: Consider increasing the precision of both the fee rate and the BPS constant to allow for more granular fee calculations. This would enable more accurate representation of small daily fees, ensuring that the compounded annual rate closely matches the intended APR.

Usual: Fixed in PR 51.

Cantina Managed: Fix verified.

3.3 Gas Optimization

3.3.1 Overly complex permit handling logic

Severity: Gas Optimization

Context: VaultRouter.sol#L208-L211

Description: The `depositWithPermit()` and `withdrawWithPermit()` functions in the `VaultRouter` contract include an unnecessary conditional check before executing a permit. The code first checks if an allowance is needed before attempting to execute the permit:

```
if (tokenIn.allowance(_msgSender(), address(this)) < permitParams.value) {
    _execPermit(tokenIn, _msgSender(), address(this), permitParams);
}
```

This adds unnecessary complexity and gas cost because:

1. It requires an additional external call to check the allowance.
2. The permit call itself would succeed or fail regardless of the current allowance.
3. If the permit succeeds, the allowance would be set to the specified value.

Recommendation: Simplify the permit handling by removing the conditional check and directly executing the permit. This will make the code more readable and gas-efficient:

```
  function depositWithPermit(
      IParaSwapAugustus augustus,
      IERC20 tokenIn,
      uint256 amount,
      uint256 minAmountToDeposit,
      uint256 minSharesReceived,
      bytes calldata swapData,
      PermitParams calldata permitParams
  )
      public
      whenNotPaused
      returns (uint256 sharesReceived)
  {
-     if (tokenIn.allowance(_msgSender(), address(this)) < permitParams.value) {
-         _execPermit(tokenIn, _msgSender(), address(this), permitParams);
-     }
+     try ERC20Permit(address(tokenIn)).permit(
+         _msgSender(),
+         address(this),
+         permitParams.value,
+         permitParams.deadline,
+         permitParams.v,
+         permitParams.r,
+         permitParams.s
+     ) {} catch {}

      return deposit(
          augustus,
          tokenIn,
          amount,
          minAmountToDeposit,
          minSharesReceived,
          swapData
      );
  }
```

Make a similar modification to the `withdrawWithPermit()` function for consistency.

Usual: PR 45 implements the recommended approach, however, the permit is applied to an incorrect address (USD0PP) in `withdrawWithPermit()`. This is fixed in PR 57. The permit is now applied correctly to the WrappedDollarVault contract.

Cantina Managed: Fix verified.

3.3.2 Redundant slippage checks for trusted operations

Severity: Gas Optimization

Context: VaultRouter.sol#L244-L249

Description: The VaultRouter contract contains unnecessary slippage and balance checks when interacting with trusted contracts. Specifically:

1. In `_convertUSD0ppToTokens()`, there is a check that compares the amount of USD0 received after unwrapping USD0PP:

```
uint256 amountUSD0 = USD0.balanceOf(address(this)) - initialUSD0Balance;
if (amountUSD0 < amountUSD0ppIn) {
    revert InsufficientUSD0Received();
}
```

2. In `withdraw()`, there is a check that verifies the amount of assets received from the vault:

```
if (IERC20(VAULT.asset()).balanceOf(address(this)) - initialTokenOutBalance < assets) {
    revert InsufficientAssetsReceived();
}
```

These checks are redundant because:

- `MINTER_USD0PP.unwrapWithCap()` is trusted and should reliably convert USD0PP to USD0 at a 1:1 ratio.

- `WrappedDollarVault` is trusted and should reliably deliver the requested assets during withdrawal.

Recommendation: Remove these redundant checks to simplify the code and reduce gas costs:

```
  function _convertUSD0ppToTokens(
      IParaSwapAugustus augustus,
      uint256 amountUSD0ppIn,
      uint256 minTokensToReceive,
      bytes calldata swapData
  )
      internal
      returns (uint256)
  {
-     uint256 initialUSD0Balance = USD0.balanceOf(address(this));

      IERC20(USD0PP).safeTransferFrom(
          _msgSender(), address(this), amountUSD0ppIn
      );

      MINTER_USD0PP.unwrapWithCap(amountUSD0ppIn);

-     uint256 amountUSD0 = USD0.balanceOf(address(this)) - initialUSD0Balance;
-     if (amountUSD0 < amountUSD0ppIn) {
-         revert InsufficientUSD0Received();
-     }

      return _executeParaswap(
-         augustus, swapData, USD0, SUSDE, amountUSD0, minTokensToReceive
+         augustus, swapData, USD0, SUSDE, amountUSD0ppIn, minTokensToReceive
      );
  }
```

```
  function withdraw(
      IParaSwapAugustus augustus,
      uint256 assets,
      uint256 minUSD0ppToReceive,
      address receiver,
      bytes calldata swapData
  )
      public
      whenNotPaused
      nonReentrant
      returns (uint256 amountUSD0pp)
  {
      if (receiver == address(0)) {
          revert NullAddress();
      }

-     uint256 initialTokenOutBalance =
-         IERC20(VAULT.asset()).balanceOf(address(this));

      VAULT.withdraw(assets, address(this), _msgSender());

-     if (
-         IERC20(VAULT.asset()).balanceOf(address(this))
-             - initialTokenOutBalance < assets
-     ) {
-         revert InsufficientAssetsReceived();
-     }

      amountUSD0pp = _convertTokensToUSD0pp(
          augustus, assets, minUSD0ppToReceive, swapData
      );

      USD0PP.safeTransfer(receiver, amountUSD0pp);
      emit Withdraw(receiver, assets, amountUSD0pp);
      return amountUSD0pp;
  }
```

Usual: Fixed in PR 47.

Cantina Managed: Fix verified.

3.4 Informational

3.4.1 Asset address stored redundantly in vault contract

Severity: Informational

Context: WrappedDollarVault.sol#L149

Description: The `WrappedDollarVault` contract stores the underlying asset address in two separate locations:

1. In the ERC4626Upgradeable parent contract's storage (set during `__ERC4626_init()`).
2. In the contract's own storage via `$.asset = IERC4626(underlyingAsset_)` in the `initialize()` function.

This redundant storage is unnecessary since the asset address can be accessed through the inherited `asset()` function, which is already used elsewhere in the contract (for example, in `_withdrawAssets()` and `_redeemShares()`).

Recommendation: Remove the redundant storage of the asset address in the contract's storage:

```
  function initialize(
      address registryContract_,
      address underlyingAsset_,
      string memory name_,
      string memory symbol_
  )
      public
      initializer
  {
      if (registryContract_ == address(0) || underlyingAsset_ == address(0)) {
          revert NullAddress();
      }
      __ReentrancyGuard_init();
      __ERC20_init(name_, symbol_);
      __ERC20Permit_init(name_);
      __ERC20Pausable_init();
      __ERC4626_init(IERC20(underlyingAsset_));
      WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
      $.registryContract = IRegistryContract(registryContract_);
      $.registryAccess = IRegistryAccess(
          $.registryContract.getContract(CONTRACT_REGISTRY_ACCESS)
      );
      $.treasury = $.registryContract.getContract(CONTRACT_YIELD_TREASURY);
-     $.asset = IERC4626(underlyingAsset_);
      $.feeRateBps = DEFAULT_FEE_RATE_BPS;
      _mint(DEAD_ADDRESS, SCALAR_ONE);
  }
```

Then replace any instances of `$.asset` with `IERC4626(address(asset()))`. This change maintains a single source of truth, reduces gas costs, and simplifies future upgrades.

Usual: Fixed in PR 41.

Cantina Managed: Fix verified.

3.4.2 Initial fee rate timestamp not set during initialization

Severity: Informational

Context: WrappedDollarVault.sol#L150

Description: In the WrappedDollarVault contract, the `$.lastFeeRateUpdateTimestamp` is not initialized during contract initialization. The contract relies on this timestamp to enforce a one-week cooling period between fee rate changes in the `setFeeRateBps()` function.

Without setting this value during initialization, the first call to `setFeeRateBps()` after deployment could potentially occur before the intended cooling period has elapsed, if called very soon after deployment. This is because the timestamp would have its default value of 0, meaning no previous update has occurred.

Recommendation: Initialize the `lastFeeRateUpdateTimestamp` field in the `initialize()` function ensuring the cooling period between fee rate updates is properly enforced starting from initialization:

```diff
  function initialize(
      address registryContract_,
      address underlyingAsset_,
      string memory name_,
      string memory symbol_
  )
      public
      initializer
  {
      if (registryContract_ == address(0) || underlyingAsset_ == address(0)) {
          revert NullAddress();
      }
      __ReentrancyGuard_init();
      __ERC20_init(name_, symbol_);
      __ERC20Permit_init(name_);
      __ERC20Pausable_init();
      __ERC4626_init(IERC20(underlyingAsset_));
      WrappedDollarVaultStorageV0 storage $ = _wrappedDollarVaultStorageV0();
      $.registryContract = IRegistryContract(registryContract_);
      $.registryAccess = IRegistryAccess(
          $.registryContract.getContract(CONTRACT_REGISTRY_ACCESS)
      );
      $.treasury = $.registryContract.getContract(CONTRACT_YIELD_TREASURY);
      $.asset = IERC4626(underlyingAsset_);
      $.feeRateBps = DEFAULT_FEE_RATE_BPS;
+     $.lastFeeRateUpdateTimestamp = block.timestamp;
      _mint(DEAD_ADDRESS, SCALAR_ONE);
  }
```

Usual: Fixed in PR 42.

Cantina Managed: Fix verified.

3.4.3 Deposit function lacks receiver parameter unlike withdraw

Severity: Informational

Context: VaultRouter.sol#L163

Description: In the `VaultRouter` contract, the `withdraw()` function allows specifying a `receiver` address to receive the withdrawn assets, but the `deposit()` function does not provide similar functionality. The `deposit()` function always uses `_msgSender()` as the receiver of vault shares. This inconsistency in function signatures limits flexibility for users who might want to deposit funds but have the resulting shares sent to a different address.

Recommendation: Consider modifying the `deposit()` function to include an optional `receiver` parameter to maintain consistency with the `withdraw()` function:

```
  function deposit(
      IParaSwapAugustus augustus,
      IERC20 tokenIn,
      uint256 amountIn,
      uint256 minTokensToReceive,
      uint256 minSharesReceived,
+     address receiver,
      bytes calldata swapData
  )
      public
      payable
      whenNotPaused
      nonReentrant
      returns (uint256 sharesReceived)
  {
+     if (receiver == address(0)) {
+         revert NullAddress();
+     }
      if (tokenIn != USDOPP && tokenIn != SUSDE) {
          revert InvalidInputToken(address(tokenIn));
      }
      uint256 tokensAmount = _convertToTokens(
          augustus, tokenIn, amountIn, minTokensToReceive, swapData
      );

-     sharesReceived = VAULT.deposit(tokensAmount, _msgSender());
+     sharesReceived = VAULT.deposit(tokensAmount, receiver);
      if (sharesReceived < minSharesReceived) {
          revert InsufficientSharesReceived();
      }
      emit IERC4626.Deposit(
-         _msgSender(), address(tokenIn), amountIn, tokensAmount
+         receiver, address(tokenIn), amountIn, tokensAmount
      );
      return sharesReceived;
  }
```

Make the same adjustment to the `depositWithPermit()` function to maintain consistency.

Usual: Fixed in PR 43.

Cantina Managed: Fix verified.

3.4.4 Incorrect event parameters in deposit function

Severity: Informational

Context: VaultRouter.sol#L188-L190

Description: The `VaultRouter` contract's `deposit()` function emits an `IERC4626.Deposit` event with incorrect parameters. The current implementation passes `address(tokenIn)` as the second parameter (which should be the owner/receiver of shares) and uses `tokensAmount` as the fourth parameter (which should be the shares received). This doesn't match the standard IERC4626 Deposit event definition:

```
event Deposit(address indexed sender, address indexed owner, uint256 assets, uint256 shares);
```

Additionally, unlike the `withdraw()` function which has a custom event, the `deposit()` function is using the ERC4626 event directly.

Recommendation: In order to maintain consistency consider creating a custom event similar to the `Withdraw` event:

```
+ // In the interface
+ event Deposit(
+     address indexed user, address indexed token, uint256 assets, uint256 shares
+) ;

  // In the deposit function
- emit IERC4626.Deposit(
-     _msgSender(), address(tokenIn), amountIn, tokensAmount
- );
+ emit Deposit(
+     _msgSender(), address(tokenIn), amountIn, sharesReceived
+ );
```

Usual: Fixed in PR 44.

Cantina Managed: Fix verified.

3.4.5 Code quality improvements

Severity: Informational

Context: VaultRouter.sol#L167-L168

Description and Recommendations:

1. Inconsistent parameter naming in `IVaultRouter` and `VaultRouter`:The parameter names in the interface and implementation don't match consistently.

```
    // In IVaultRouter.sol
    function deposit(
        IParaSwapAugustus augustus,
        IERC20 tokenIn,
        uint256 amountIn,
  -     uint256 minSUSDeToReceive,
  +     uint256 minTokensToReceive,
        uint256 minSharesReceived,
        bytes calldata swapData
    )
        external
        payable
        returns (uint256 sharesReceived);
```

2. Inconsistent naming in the Withdraw event: The event parameter names should match the terminology used in the functions:

```
    event Withdraw(
  -     address indexed user,
  +     address indexed receiver,
  -     uint256 amountSUSDe,
  +     uint256 assets,
        uint256 amountUSD0pp
    );
```

3. Standardize naming conventions across all functions: For consistency across all functions, standardize parameter names for both deposit and withdraw operations:

```
    function depositWithPermit(
        IParaSwapAugustus augustus,
        IERC20 tokenIn,
  -     uint256 amount,
  +     uint256 amountIn,
  -     uint256 minAmountToDeposit,
  +     uint256 minTokensToReceive,
        uint256 minSharesReceived,
        bytes calldata swapData,
        PermitParams calldata permitParams
    )
        external
        returns (uint256 sharesReceived);
```

```
   function deposit(
       IParaSwapAugustus augustus,
       IERC20 tokenIn,
       uint256 amount,
       uint256 minTokensToReceive,
-      uint256 minSharesReceived,
+      uint256 minSharesToReceive,
       bytes calldata swapData
   )
```

Usual: Fixed in PR 48.

Cantina Managed: Fix verified.