CANTINA

Alto Money

Security Review

Cantina Managed review by:
Devtooligan, Lead Security Researcher
Phaze, Lead Security Researcher

December 8, 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

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 level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

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 are 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

Alto is a decentralized credit protocol built around DUSD, focused keeping risk contained by isolating markets, using a single stablecoin and partial liquidations.

From Nov 23rd to Nov 25th the Cantina team conducted a review of Alto Money on commit hash 75fd6ea2. The team identified a total of **8** 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	0	0	0
Informational	7	4	3
Total	**8**	**5**	**3**

2.1 Scope

The security review had the following components in scope for Alto Money on commit hash 75fd6ea2:

```
contracts/usm
├── DUSDUsm.sol
├── FixedFeeStrategy.sol
├── FixedPriceStrategy.sol
└── Usm.sol
```

3 Findings

3.1 Low Risk

3.1.1 Incomplete state reset during contract seizure

Severity: Low Risk

Context: Usm.sol#L162-L192

Description: The `seize()` function in the USM contract does not reset the `_accruedFees` state variable when the contract is seized. During seizure, the function resets several state variables to reflect the liquidated state but leaves accrued fees unchanged:

```
function seize() external notSeized onlyRole(LIQUIDATOR_ROLE) returns (uint256) {
    _isSeized = true;
    _currentExposure = 0;        // Reset to 0
    _updateExposureCap(0);       // Reset to 0

    // Transfer underlying assets to treasury
    uint256 underlyingBalance = IERC20(UNDERLYING_ASSET).balanceOf(address(this));
    if (underlyingBalance > 0) {
        IERC20(UNDERLYING_ASSET).safeTransfer(_stableTokenTreasury, underlyingBalance);
    }

    // _accruedFees not reset
    emit Seized(msg.sender, _stableTokenTreasury, underlyingBalance, stableTokenMinted);
    return underlyingBalance;
}
```

This creates an accounting mismatch where accrued fees remain recorded while the contract lacks sufficient stable token balance to distribute them. If `distributeFeesToTreasury()` were called after seizure, it would attempt to transfer stable tokens equal to the accrued fees but would likely revert due to insufficient balance, since the underlying collateral was already transferred during seizure.

Recommendation: Consider resetting `_accruedFees` during seizure for consistency with other state variables and to prevent accounting mismatches:

```
  function seize() external notSeized onlyRole(LIQUIDATOR_ROLE) returns (uint256) {
      _isSeized = true;
      _currentExposure = 0;
+     _accruedFees = 0;
      _updateExposureCap(0);

      uint256 stableTokenMinted = _getCurrentlyMintedStableTokenByUsm();
      uint256 underlyingBalance = IERC20(UNDERLYING_ASSET).balanceOf(address(this));
      if (underlyingBalance > 0) {
          IERC20(UNDERLYING_ASSET).safeTransfer(_stableTokenTreasury, underlyingBalance);
      }

      emit Seized(msg.sender, _stableTokenTreasury, underlyingBalance,
       ↪  stableTokenMinted);
      return underlyingBalance;
  }
```

This ensures complete state cleanup during the seizure process and prevents potential reverts from subsequent fee distribution attempts.

Alto: Fixed in PR 389.

Cantina Managed: Fix verified.

3.2 Informational

3.2.1 Missing validation in fee strategy update function

Severity: Informational

Context: Usm.sol#L467-L475

Description: The `_updateFeeStrategy()` function in the USM contract lacks input validation when setting a new fee strategy address. Unlike the corresponding `_updatePriceStrategy()` function, which validates that the new price strategy is compatible with the underlying asset, the fee strategy update accepts any address without verification.

```
function _updateFeeStrategy(address feeStrategy) internal {
    address oldFeeStrategy = _feeStrategy;
    _feeStrategy = feeStrategy;
    emit FeeStrategyUpdated(oldFeeStrategy, feeStrategy);
}
```

While fee strategies may be less tightly coupled to specific underlying assets than price strategies, the absence of any validation could allow the configuration of incompatible or unwanted fee strategy contracts. This could lead to unexpected behavior in fee calculations throughout the system.

Recommendation: Consider adding validation to ensure the fee strategy contract implements the expected interface and behaves correctly. This could include:

```
function _updateFeeStrategy(address feeStrategy) internal {
    if (feeStrategy != address(0)) {
        // Validate that the contract implements the expected interface
        require(
            IUsmFeeStrategy(feeStrategy).supportsInterface(type(IUsmFeeStrategy).interfa⌋
            ↪   ceId),
            "INVALID_FEE_STRATEGY"
        );

        // Additional validation could include testing basic fee calculations
        // to ensure the strategy behaves as expected
    }

    address oldFeeStrategy = _feeStrategy;
    _feeStrategy = feeStrategy;
    emit FeeStrategyUpdated(oldFeeStrategy, feeStrategy);
}
```

Alto: Acknowledged. We will consider adding this in the future as currently the rest of our codebase does not have interface checks.

Cantina Managed: Acknowledged.

3.2.2 Optional fee calculation may enable arbitrage opportunities

Severity: Informational

Context: Usm.sol#L248-L256

Description: The `getAssetAmountForBuyAsset()` function (and similar functions throughout the USM contract) makes fee calculations optional based on whether a fee strategy is configured. When `_feeStrategy` is set to `address(0)`, fees are effectively zero:

```
bool withFee = _feeStrategy != address(0);
uint256 grossAmount = withFee
    ? IUsmFeeStrategy(_feeStrategy).getGrossAmountFromTotalBought(maxStableTokenAmount)
    : maxStableTokenAmount;
// ...
uint256 finalFee = withFee ? IUsmFeeStrategy(_feeStrategy).getBuyFee(finalGrossAmount) :
↪   0;
```

Zero-fee configurations may create arbitrage opportunities where sophisticated actors can exploit the absence of trading costs to extract value from the system through rapid buy/sell cycles or price discrepancies across different venues.

Recommendation: Consider implementing a minimum fee mechanism to maintain system stability even when no fee strategy is configured:

```
function getAssetAmountForBuyAsset(uint256 maxStableTokenAmount)
    external
    view
    returns (uint256, uint256, uint256, uint256)
{
    uint256 grossAmount = _feeStrategy != address(0)
        ? IUsmFeeStrategy(_feeStrategy).getGrossAmountFromTotalBought(maxStableTokenAmou
        ↪   nt)
        : maxStableTokenAmount;

    uint256 assetAmount =
    ↪   IUsmPriceStrategy(priceStrategy).getStableTokenPriceInAsset(grossAmount, false);
    uint256 finalGrossAmount =
    ↪   IUsmPriceStrategy(priceStrategy).getAssetPriceInStableToken(assetAmount, true);

    uint256 finalFee = _feeStrategy != address(0)
        ? IUsmFeeStrategy(_feeStrategy).getBuyFee(finalGrossAmount)
        : finalGrossAmount * MINIMUM_FEE_BPS / 10000; // e.g., 1 basis point minimum

    return (assetAmount, finalGrossAmount + finalFee, finalGrossAmount, finalFee);
}
```

This approach ensures that some level of fee protection remains in place regardless of the fee strategy configuration, reducing the potential for exploitation while maintaining operational flexibility.

Alto: Acknowledged. We will be making sure to never add the no-fee USM and will always thoroughly discuss fees with our risk mangement.

Cantina Managed: Acknowledged.

3.2.3 Front-running vulnerability in signature-based authorization

Severity: Informational

Context: AuthUpgradeable.sol#L39-L62

Description: The `setAuthorizationWithSignature()` function in `AuthUpgradeable.sol` is vulnerable to front-running attacks when used as part of bundled transactions. If a user intends to bundle this function call with other operations, an attacker could observe the pending transaction and call `setAuthorizationWithSignature()` directly with the same parameters, causing the original bundled transaction to fail due to nonce mismatch.

```
function setAuthorizationWithSignature(Authorization memory authData, Signature calldata
↪   sigData) external {
    if (block.timestamp > authData.deadline) {
        revert AuthSignatureExpired();
    }
    if (authData.nonce != nonce[authData.owner]++) {  // Nonce incremented here
        revert AuthInvalidNonce();
    }
    // ... signature validation and authorization setting
}
```

When the front-runner calls this function, it increments the nonce for `authData.owner`, causing the subsequent bundled transaction to revert when it attempts to use the same nonce.

The risk is limited as there is no direct financial incentive for attackers beyond temporarily disrupting operations. Additionally, bundling systems like Bundler3 provide a `skipRevert` option that can mitigate transaction failures from individual call reverts.

Recommendation: Consider using the bundler's `skipRevert` functionality when bundling authorization calls to gracefully handle potential front-running scenarios. This allows the bundled transaction to continue executing even if the authorization call fails due to front-running, preventing disruption of the overall operation.

Alto: Acknowledged. We will set `skipRevert` to true when dealing with bundled signatures.

Cantina Managed: Acknowledged.

3.2.4 Manual decimal specification may lead to configuration errors

Severity: Informational

Context: FixedPriceStrategy.sol#L46-L47

Description: The `FixedPriceStrategy` constructor requires manual specification of the underlying asset's decimal places through the `underlyingAssetDecimals` parameter. This approach introduces the risk of configuration errors where the provided decimal count doesn't match the actual decimals of the underlying asset token contract.

```
constructor(uint256 initialPriceRatio, address underlyingAsset, uint8
↪   underlyingAssetDecimals, address admin) {
    // ...
    UNDERLYING_ASSET = underlyingAsset;
    UNDERLYING_ASSET_DECIMALS = underlyingAssetDecimals;
    _underlyingAssetUnits = 10 ** underlyingAssetDecimals;
}
```

If the manually provided decimals value is incorrect, all price calculations in `getAssetPriceInStableToken()` and `getStableTokenPriceInAsset()` will be wrong, potentially leading to significant pricing errors in the USM system.

Recommendation: Consider fetching the decimal count directly from the underlying asset token contract to eliminate this configuration risk:

```
+ import {IERC20Metadata} from
↪   "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

- constructor(uint256 initialPriceRatio, address underlyingAsset, uint8
↪   underlyingAssetDecimals, address admin) {
+ constructor(uint256 initialPriceRatio, address underlyingAsset, address admin) {
    require(initialPriceRatio > 0, "INVALID_PRICE_RATIO");
    require(admin != address(0), "ZERO_ADDRESS_NOT_VALID");
    _grantRole(DEFAULT_ADMIN_ROLE, admin);
    _grantRole(CONFIGURATOR_ROLE, admin);

    priceRatio = initialPriceRatio;
    UNDERLYING_ASSET = underlyingAsset;
-   UNDERLYING_ASSET_DECIMALS = underlyingAssetDecimals;
+   UNDERLYING_ASSET_DECIMALS = IERC20Metadata(underlyingAsset).decimals();
    _underlyingAssetUnits = 10 ** UNDERLYING_ASSET_DECIMALS;
  }
```

This approach ensures the decimal count always matches the actual token contract, reducing deployment risks and improving system reliability.

Alto: Fixed in commit 77b9be17.

Cantina Managed: Fix verified.

3.2.5 Insufficient minimum amount validation in asset purchase

Severity: Informational

Context: Usm.sol#L115

Finding Description: The `_buyAsset()` function lacks explicit validation that the calculated asset amount meets the user-specified minimum requirement. While the function accepts a `minAmount` parameter representing the minimum acceptable asset purchase, it does not verify that `finalAssetAmount` satisfies this constraint before executing the transfer.

The calculation flow in `_calculateStableTokenAmountForBuyAsset()` proceeds as follows:

1. Calculate the gross stable token cost for `minAmount` using `getAssetPriceInStableToken()` with rounding up.
2. Add fees to determine total stable tokens required.
3. Recalculate the actual asset amount using `getStableTokenPriceInAsset()` with rounding down.

The protocol team notes that the rounding directions mathematically guarantee `finalAssetAmount >= minAmount` under the assumption that price and fee strategies are correctly implemented. The upward rounding when calculating cost and downward rounding when converting back to assets should preserve this inequality.

However, this guarantee depends entirely on the correctness of external strategy contracts. If a price or fee strategy contains implementation errors, returns unexpected values, or behaves maliciously, the implicit guarantee breaks down. The lack of explicit validation means the contract would proceed with transfers that violate user expectations.

Recommendation: Consider adding an explicit check to validate the minimum amount requirement:

```
  function _buyAsset(address originator, uint256 minAmount, address receiver) internal
  ↪   returns (uint256, uint256) {
      (uint256 assetAmount, uint256 stableTokenSold, uint256 grossAmount, uint256 fee) =
          _calculateStableTokenAmountForBuyAsset(minAmount);

      _beforeBuyAsset(originator, assetAmount, receiver);

      require(assetAmount > 0, "INVALID_AMOUNT");
+     require(assetAmount >= minAmount, "INSUFFICIENT_OUTPUT_AMOUNT");
      require(_currentExposure >= assetAmount,
      ↪   "INSUFFICIENT_AVAILABLE_EXOGENOUS_ASSET_LIQUIDITY");

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

Alto: Fixed in PR 388.

Cantina Managed: Fix verified.

3.2.6 Missing events for configuration tracking in fee and price strategies

Severity: Informational

Context: FixedFeeStrategy.sol#L42-L49, FixedPriceStrategy.sol#L51-L55

Description: The `FixedFeeStrategy` and `FixedPriceStrategy` contracts lack event emissions when critical configuration parameters are updated. This makes it difficult to track changes to these important system parameters.

In `FixedFeeStrategy.sol`, the `setFee()` function updates buy and sell fees without emitting an event:

```
function setFee(uint256 buyFee, uint256 sellFee) external onlyRole(CONFIGURATOR_ROLE) {
    require(buyFee < MAXIMUM_FEE_PERCENT, "INVALID_BUY_FEE");
    require(sellFee < MAXIMUM_FEE_PERCENT, "INVALID_SELL_FEE");
    require(buyFee > 0 || sellFee > 0, "MUST_HAVE_ONE_NONZERO_FEE");
    _buyFee = buyFee;
    _sellFee = sellFee;
    // Missing event emission
}
```

Similarly, in `FixedPriceStrategy.sol`, the `setPriceRatio()` function updates the price ratio without emitting an event:

```
function setPriceRatio(uint256 newPriceRatio) external onlyRole(CONFIGURATOR_ROLE) {
    require(newPriceRatio > 0, "INVALID_PRICE_RATIO");
    priceRatio = newPriceRatio;
    // Missing event emission
}
```

Recommendation: Consider adding events to both functions to improve tracking and monitoring capabilities:

```
// In FixedFeeStrategy.sol
event FeesUpdated(uint256 buyFee, uint256 sellFee);

function setFee(uint256 buyFee, uint256 sellFee) external onlyRole(CONFIGURATOR_ROLE) {
    require(buyFee < MAXIMUM_FEE_PERCENT, "INVALID_BUY_FEE");
    require(sellFee < MAXIMUM_FEE_PERCENT, "INVALID_SELL_FEE");
    require(buyFee > 0 || sellFee > 0, "MUST_HAVE_ONE_NONZERO_FEE");
    _buyFee = buyFee;
    _sellFee = sellFee;
    emit FeesUpdated(buyFee, sellFee);
}

// In FixedPriceStrategy.sol
event PriceRatioUpdated(uint256 oldPriceRatio, uint256 newPriceRatio);

function setPriceRatio(uint256 newPriceRatio) external onlyRole(CONFIGURATOR_ROLE) {
    require(newPriceRatio > 0, "INVALID_PRICE_RATIO");
    uint256 oldPriceRatio = priceRatio;
    priceRatio = newPriceRatio;
    emit PriceRatioUpdated(oldPriceRatio, newPriceRatio);
}
```

Alto: Fixed in PR 390.

Cantina Managed: Fix verified.

3.2.7 Code quality improvements

Severity: Informational

Context: Usm.sol#L214

Description/Recommendations: *(See each case below)*

1. Use `safeTransfer` for fee distribution.

   The `distributeFeesToTreasury()` function uses the standard `transfer()` method instead of `safeTransfer()` for token transfers:

   ```
   IERC20(STABLE_TOKEN).transfer(_stableTokenTreasury, accruedFees);
   ```

   While this is not a security concern given that the stable token is a well-behaved token created by the team, using `safeTransfer()` follows best practices and ensures consistent error handling across the codebase:

   ```
   - IERC20(STABLE_TOKEN).transfer(_stableTokenTreasury, accruedFees);
   + IERC20(STABLE_TOKEN).safeTransfer(_stableTokenTreasury, accruedFees);
   ```

2. Correct price ratio documentation.

   The comment for `priceRatio` in `FixedPriceStrategy.sol` incorrectly describes the units:

   ```
   /// @dev The price ratio from underlying asset to stability token (expressed in
   ↪  WAD)
   uint256 public priceRatio;
   ```

   The price ratio is actually expressed in the underlying asset's units rather than WAD. Update the documentation for accuracy:

   ```
   - /// @dev The price ratio from underlying asset to stability token (expressed in
   ↪  WAD)
   + /// @dev The price ratio from underlying asset to stability token (expressed in
   ↪  stable token units)
   ```

Alto: Fixed in commit 10a5d9df.

Cantina Managed: Fix verified.