Alto Money v1

Security Review

Cantina Solo review by:
Phaze, Lead Security Researcher

December 9, 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 security review is 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 the review 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 a 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 30th to Dec 3rd the security researchers conducted a review of v1 on commit hash 2cae67d9. A total of **9** issues were identified:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	2	1	1
Gas Optimizations	1	0	1
Informational	6	3	3
Total	**9**	**4**	**5**

2.1 Scope

The security review had the following components in scope for v1 on commit hash 2cae67d9:

```
contracts
├── lending
│   └── leverage
│       └── AltoLeverageSwapper.sol
├── oracles
│   ├── AltoERC4626RatioChainlinkOracle
│   │   ├── AltoERC4626RatioChainlinkOracle.sol
│   │   └── implementations
│   │       ├── ERC4626RatioChainlinkOracleMETH.sol
│   │       ├── ERC4626RatioChainlinkOracleRETH.sol
│   │       ├── ERC4626RatioChainlinkOracleSUSDE.sol
│   │       └── ERC4626RatioChainlinkOracleWSTETH.sol
│   ├── AltoMultiChainlinkLendingOracle.sol
│   ├── AltoRewardsOracle.sol
│   └── modules
│       ├── ModuleChainlink.sol
│       ├── ModuleCurve.sol
│       ├── ModuleERC4626Ratio.sol
│       └── ModuleUniswap
│           ├── FullMath.sol
│           ├── ModuleUniswap.sol
│           └── OracleMath.sol
├── tokens
│   └── DUSD.sol
└── vesting
    └── AltoVesting.sol
```

3 Findings

3.1 Low Risk

3.1.1 Potential arithmetic overflow in Chainlink price conversion for large input amounts

Severity: Low Risk

Context: ModuleChainlink.sol#L22-L45

Description: The `readChainlinkFeed()` function in `ModuleChainlink` performs multiplication operations that could overflow when processing very large input amounts. While the protocol's typical usage patterns involve small values (around 1 unit of the underlying asset), the function is public and could be used externally with larger amounts, potentially causing unexpected reverts.

The `readChainlinkFeed()` function converts prices between different decimal representations by performing multiplications before divisions:

```
function readChainlinkFeed(uint256 quoteAmount, ChainlinkNode memory config) public view
↪  returns (uint256) {
    // ... validation ...

    uint256 price = uint256(ratio);
    uint256 powFeed = 10 ** config.feed.decimals();
    uint256 powIn = 10 ** config.inDecimals;
    uint256 powOut = 10 ** config.outDecimals;

    if (config.direction == ConversionDirection.IN_TO_OUT) {
        // quoteAmount = quoteAmount * ratio / 10^feedDecimals * 10^outDecimals /
        ↪  10^inDecimals
        quoteAmount = FixedPointMath.divideWithRounding(quoteAmount, price * powOut,
        ↪  powFeed * powIn, false);
    } else {
        // quoteAmount = quoteAmount * 10^feedDecimals / ratio * 10^outDecimals /
        ↪  10^inDecimals
        quoteAmount = FixedPointMath.divideWithRounding(quoteAmount, powFeed * powOut,
        ↪  price * powIn, false);
    }

    return quoteAmount;
}
```

The `divideWithRounding()` function uses standard Solidity multiplication and division, which can overflow at 2^256. For a typical ETH oracle configuration:

- Feed decimals: 8.
- Output decimals (USD precision): 36.
- Input amount: ~1e18 (1 ETH).
- Price: ~1e11 (at $3,000 per ETH).

The numerator calculation becomes: `quoteAmount * price * powOut = 1e18 * 1e11 * 1e36 = 1e65`.

This approaches the uint256 maximum of ~1e77, leaving a margin of only 1e12. The function would overflow when the input amount reaches approximately 1e12 ether (depending on the current ETH price), or similar values for other assets.

While the protocol's oracles typically call this function with values around 1 unit of the underlying asset (as seen in `AltoMultiChainlinkLendingOracle.getPrice()` and `AltoERC4626RatioChainlinkOracle.getPrice()`), the function is public and could be used by external contracts or integrators with larger amounts.

Additionally, the parameter is named `quoteAmount`, which is misleading. In standard trading terminology, the "quote" is the output amount (e.g., USD), while the input should be called the "base" or "asset" amount.

Recommendation: Consider implementing one of the following approaches to make the function more robust:

1. Use 512-bit precision arithmetic.

The codebase already includes Uniswap's `FullMath` library, which handles intermediate overflows using 512-bit precision. Consider using it for the price conversion:

```
  function readChainlinkFeed(uint256 quoteAmount, ChainlinkNode memory config) public
  ↪  view returns (uint256) {
      // ... validation ...

      uint256 price = uint256(ratio);
      uint256 powFeed = 10 ** config.feed.decimals();
      uint256 powIn = 10 ** config.inDecimals;
      uint256 powOut = 10 ** config.outDecimals;

      if (config.direction == ConversionDirection.IN_TO_OUT) {
-         quoteAmount = FixedPointMath.divideWithRounding(quoteAmount, price * powOut,
↪  powFeed * powIn, false);
+         quoteAmount = _mulDiv(quoteAmount * price, powOut, powFeed * powIn);
      } else {
-         quoteAmount = FixedPointMath.divideWithRounding(quoteAmount, powFeed * powOut,
↪  price * powIn, false);
+         quoteAmount = _mulDiv(quoteAmount * powFeed, powOut, price * powIn);
      }

      return quoteAmount;
  }
```

2. Pre-calculate decimal adjustments.

Reduce the numerator size by pre-calculating the net decimal adjustment:

```
function readChainlinkFeed(uint256 baseAmount, ChainlinkNode memory config) public view
↪  returns (uint256) {
    // ... validation ...

    uint256 price = uint256(ratio);
    uint256 feedDecimals = config.feed.decimals();

    // Calculate net decimal adjustment
    (uint256 numeratorPow, uint256 denominatorPow) =
        _calculateDecimalAdjustment(config.outDecimals, config.inDecimals, feedDecimals);

    if (config.direction == ConversionDirection.IN_TO_OUT) {
        baseAmount = baseAmount * price * numeratorPow / denominatorPow;
    } else {
        baseAmount = baseAmount * numeratorPow / (price * denominatorPow);
    }

    return baseAmount;
}

function _calculateDecimalAdjustment(
    uint256 outDecimals,
    uint256 inDecimals,
    uint256 feedDecimals
) internal pure returns (uint256 numeratorPow, uint256 denominatorPow) {
    int256 exponentDiff;

    if (direction == ConversionDirection.IN_TO_OUT) {
        // For IN_TO_OUT: outDecimals / (feedDecimals + inDecimals)
        // Net exponent: outDecimals - feedDecimals - inDecimals
        exponentDiff = int256(outDecimals) - int256(feedDecimals) - int256(inDecimals);
    } else {
        // For OUT_TO_IN: (feedDecimals + outDecimals) / inDecimals
        // Net exponent: feedDecimals + outDecimals - inDecimals
        exponentDiff = int256(feedDecimals) + int256(outDecimals) - int256(inDecimals);
    }
```

```
    numeratorPow = 1;
    denominatorPow = 1;

    if (exponentDiff > 0) {
        numeratorPow = 10 ** uint256(exponentDiff);
    } else if (exponentDiff < 0) {
        denominatorPow = 10 ** uint256(-exponentDiff);
    }
}
```

This approach would reduce the numerator for ETH from 1e65 to approximately 1e36, providing significantly more headroom before overflow (up to ~1e41 ether).

Alto: Fixed in PR 408.

Phaze: Fix verified.

3.1.2 Oracle price feeds assume 1:1 peg for wrapped Bitcoin assets

Severity: Low Risk

Context: EthereumMainnetOracleDeployer.sol#L142-L190

Summary: The oracle deployments for cbBTC and tBTC assume a 1:1 exchange rate with Bitcoin without accounting for potential depegging. Both assets have direct Chainlink USD price feeds available that would eliminate this assumption and provide accurate market-based pricing. This contrasts with the WBTC oracle which properly uses a dedicated WBTC/BTC ratio feed to account for potential deviations from the peg.

Description: The protocol deploys oracles for three wrapped Bitcoin assets: WBTC, cbBTC, and tBTC. However, these deployments handle the wrapped asset to BTC relationship inconsistently.

WBTC Oracle (Proper Implementation):

The `deployWbtcOracle()` function properly accounts for the WBTC/BTC ratio by chaining two Chainlink feeds:

```
function deployWbtcOracle(address owner) public returns (address) {
    ChainlinkNode[] memory chainlinkNodes = new ChainlinkNode[](2);

    // First feed: WBTC/BTC ratio
    chainlinkNodes[0] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_WBTC_BTC_RATIO_FEED),
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 26 hours,
        inDecimals: 8,
        outDecimals: 8
    });

    // Second feed: BTC/USD price
    chainlinkNodes[1] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_BTC_USD_PRICE_FEED),
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 2 hours,
        inDecimals: 8,
        outDecimals: 46
    });

    return address(new AltoMultiChainlinkLendingOracle(owner, chainlinkNodes));
}
```

This approach correctly prices WBTC as: `WBTC Price = (WBTC/BTC ratio) × (BTC/USD price)`.

cbBTC and tBTC Oracles (Incomplete Implementation):

Both `deployCbbtcOracle()` and `deployTbtcOracle()` assume a 1:1 ratio with BTC by using only the BTC/USD price feed:

```
function deployCbbtcOracle(address owner) public returns (address) {
    ChainlinkNode[] memory chainlinkNodes = new ChainlinkNode[](1);

    chainlinkNodes[0] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_BTC_USD_PRICE_FEED),
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 2 hours,
        inDecimals: 8,
        outDecimals: 46
    });

    return address(new AltoMultiChainlinkLendingOracle(owner, chainlinkNodes));
}

function deployTbtcOracle(address owner) public returns (address) {
    ChainlinkNode[] memory chainlinkNodes = new ChainlinkNode[](1);

    chainlinkNodes[0] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_BTC_USD_PRICE_FEED),
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 2 hours,
        inDecimals: 18,
        outDecimals: 36
    });

    return address(new AltoMultiChainlinkLendingOracle(owner, chainlinkNodes));
}
```

Both implementations effectively price the assets as: `Asset Price = BTC/USD price` (assuming 1:1 peg).

However, Chainlink provides direct USD price feeds for both assets that reflect their actual market values:

- cbBTC/USD feed: `0x2665701293fCbEB223D11A08D826563EDcCE423A`.
- tBTC/USD feed: `0x8350b7De6a6a2C1368E7D4Bd968190e13E354297`.

Risk of 1:1 Peg Assumption: Wrapped Bitcoin assets have historically maintained close to 1:1 pegs with Bitcoin, but depegging events can occur due to:

- Custodial issues with the backing reserves.
- Smart contract vulnerabilities.
- Regulatory actions affecting the issuer.
- Market liquidity problems.

When a wrapped asset depegs, using only the BTC/USD price would result in either overvaluing or undervaluing the collateral.

In a lending protocol, overvalued collateral is particularly dangerous as it allows users to borrow more than their collateral is actually worth, potentially leading to bad debt accumulation.

Impact Explanation: The impact is high. If cbBTC or tBTC were to depeg from Bitcoin, the lending protocol would incorrectly value the collateral, potentially allowing:

- Users to over-borrow against depegged assets.
- Liquidations to fail due to overvalued collateral.
- Accumulation of bad debt in the protocol.

Likelihood Explanation: The likelihood is low. While wrapped Bitcoin assets have generally maintained strong pegs historically, depegging events have occurred in crypto markets (though not specifically with these assets). The risk increases during market stress or if issues arise with the asset issuers.

Recommendation: Use the direct Chainlink USD price feeds for both cbBTC and tBTC instead of assuming a 1:1 peg with Bitcoin:

- For cbBTC:

```
function deployCbbtcOracle(address owner) public returns (address) {
    ChainlinkNode[] memory chainlinkNodes = new ChainlinkNode[](1);

    chainlinkNodes[0] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_CBBTC_USD_PRICE_FEED), // cbBTC/USD
        ↪  feed
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 2 hours,
        inDecimals: 8,
        outDecimals: 46
    });

    return address(new AltoMultiChainlinkLendingOracle(owner, chainlinkNodes));
}
```

- For tBTC:

```
function deployTbtcOracle(address owner) public returns (address) {
    ChainlinkNode[] memory chainlinkNodes = new ChainlinkNode[](1);

    chainlinkNodes[0] = ChainlinkNode({
        feed: AggregatorV3Interface(CHAINLINK_TBTC_USD_PRICE_FEED), // tBTC/USD
        ↪  feed
        direction: ConversionDirection.IN_TO_OUT,
        stalePeriod: 2 hours,
        inDecimals: 18,
        outDecimals: 36
    });

    return address(new AltoMultiChainlinkLendingOracle(owner, chainlinkNodes));
}
```

These direct price feeds provide accurate market-based pricing that reflects actual trading values rather than assuming maintained pegs with Bitcoin.

Alto: Acknowledged.

Phaze: Acknowledged.

3.2 Gas Optimization

3.2.1 Chainlink feed decimals could be cached to reduce external calls

Severity: Gas Optimization

Context: ModuleERC4626Ratio.sol#L58-L60

Description: The `readChainlinkFeed()` function in `ModuleChainlink.sol` calls `config.feed.decimals()` on every invocation to retrieve the Chainlink feed's decimal precision:

```
function readChainlinkFeed(uint256 quoteAmount, ChainlinkNode memory config) public view
↪  returns (uint256) {
    (uint80 roundId, int256 ratio,, uint256 updatedAt, uint80 answeredInRound) =
    ↪  config.feed.latestRoundData();
    if (ratio <= 0 || roundId > answeredInRound || block.timestamp - updatedAt >
    ↪  config.stalePeriod) {
        revert ModuleChainlinkInvalidRate();
    }

    // External call to get decimals
    uint256 price = uint256(ratio);
    uint256 powFeed = 10 ** config.feed.decimals();  // <-- External call on every read
    uint256 powIn = 10 ** config.inDecimals;
    uint256 powOut = 10 ** config.outDecimals;
```

```
    // ... price calculation ...
}
```

This external call to retrieve the feed's decimals occurs on every price read operation. Since Chainlink feed decimals are immutable and never change after deployment, this value could be cached in the `ChainlinkNode` configuration struct and validated once during initialization.

Recommendation: Store the feed decimals directly in the `ChainlinkNode` struct and validate it during configuration setup:

- Update the ChainlinkNode struct:

```
struct ChainlinkNode {
    AggregatorV3Interface feed;
    ConversionDirection direction;
    uint32 stalePeriod;
    uint8 inDecimals;
    uint8 outDecimals;
    uint8 feedDecimals;  // Add cached feed decimals
}
```

- Validate the cached value matches the actual feed:

```
function _validateChainlinkParams(ChainlinkNode memory config) internal view {
    if (address(config.feed) == address(0) || config.stalePeriod == 0) {
        revert ModuleChainlinkInvalidInput();
    }

    // Validate that cached feedDecimals matches actual feed decimals
    if (config.feedDecimals != config.feed.decimals()) {
        revert ModuleChainlinkInvalidInput();
    }

    // Bound decimals to prevent overflow and DoS via extreme values
    if (config.inDecimals + config.outDecimals > 54) {
        revert ModuleChainlinkInvalidInput();
    }
}
```

- Use the cached value in readChainlinkFeed:

```
function readChainlinkFeed(uint256 quoteAmount, ChainlinkNode memory config)
↪   public view returns (uint256) {
    (uint80 roundId, int256 ratio,, uint256 updatedAt, uint80 answeredInRound) =
    ↪   config.feed.latestRoundData();
    if (ratio <= 0 || roundId > answeredInRound || block.timestamp - updatedAt >
    ↪   config.stalePeriod) {
        revert ModuleChainlinkInvalidRate();
    }

    uint256 price = uint256(ratio);
    uint256 powFeed = 10 ** config.feedDecimals;  // Use cached value instead of
    ↪   external call
    uint256 powIn = 10 ** config.inDecimals;
    uint256 powOut = 10 ** config.outDecimals;

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

This optimization eliminates one external call per oracle read, reducing gas costs for all contracts that consume these oracle prices. The one-time validation during initialization ensures the cached value remains accurate.

Alto: Acknowledged.

Phaze: Acknowledged.

3.3 Informational

3.3.1 User-supplied initial parameters increase risk of misconfiguration in ratio tracking

Severity: Informational

Context: ModuleERC4626Ratio.sol#L51-L57

Description: The `ModuleERC4626Ratio` contract relies on user-supplied `initialRatio` and `initialTimestamp` parameters during initialization to establish the baseline for tracking ratio growth over time. The contract validates these parameters in `_validateERC4626RatioParams()`, checking that `initialRatio` is non-zero and `initialTimestamp` is not in the future. However, there is no validation that these values accurately reflect the actual on-chain state at the time of initialization.

The `_computeMaxRatio()` function uses these parameters to calculate the maximum allowed ratio growth:

```
function _computeMaxRatio(ERC4626RatioNode memory _erc4626RatioNode) internal view
↪  returns (uint256) {
    uint256 timePassed = block.timestamp - _erc4626RatioNode.initialTimestamp;
    uint256 maxGrowthPerYear = _erc4626RatioNode.initialRatio *
    ↪  _erc4626RatioNode.maxYearlyGrowthRatioWad;
    uint256 maxAllowedGrowthRatio = maxGrowthPerYear * timePassed / MATH_PRECISION /
    ↪  SECONDS_PER_YEAR;
    return _erc4626RatioNode.initialRatio + maxAllowedGrowthRatio;
}
```

If the `initialRatio` or `initialTimestamp` are set incorrectly, an `initialRatio` set too low or an initialTimestamp' in the past would allow the actual ratio to grow beyond intended limits before the cap takes effect. Conversely, if setting it too low would unnecessarily restrict the reported ratio.

Since the contract already implements `_getRatioOfWrappedAssetToUnderlyingAsset()` to fetch the current ratio from the underlying vault, these values could be determined programmatically at initialization time rather than relying on manual input.

Recommendation: Consider modifying the initialization process to automatically fetch the initial parameters rather than accepting them as user input. This would involve moving the `_getRatioOfWrappedAssetToUnderlyingAsset()` function from the child contracts to the `ModuleERC4626Ratio` contract and calling it during initialization:

```
  function _validateERC4626RatioParams(ERC4626RatioNode memory _erc4626RatioNode)
  ↪  internal view {
-     if (_erc4626RatioNode.initialRatio == 0) {
-         revert ModuleERC4626RatioInvalidInput();
-     }
-     if (_erc4626RatioNode.initialTimestamp > block.timestamp) {
-         revert ModuleERC4626RatioInvalidInput();
-     }
      if (_erc4626RatioNode.maxYearlyGrowthRatioWad == 0) {
          revert ModuleERC4626RatioInvalidInput();
      }
  }

+  function _initializeERC4626RatioParams(
+      uint256 maxYearlyGrowthRatioWad
+  ) internal {
+      erc4626RatioNode = ERC4626RatioNode({
+          initialRatio: _getRatioOfWrappedAssetToUnderlyingAsset(),
+          initialTimestamp: block.timestamp,
+          maxYearlyGrowthRatioWad: maxYearlyGrowthRatioWad
+      });
+  }
```

This approach would reduce the risk of misconfiguration through human errors.

Alto: Acknowledged.

Phaze: Acknowledged.

3.3.2 Manual initial price in TWAP initialization increases risk of misconfiguration

Severity: Informational

Context: ModuleERC4626Ratio.sol#L51-L57

Description: The `AltoRewardsOracle` contract's `startTwapPeriod()` function requires the owner to manually provide an initial price when starting the TWAP period. However, the contract already contains the necessary infrastructure to fetch the current price from Uniswap and Chainlink oracles in the `updatePrice()` function.

The `startTwapPeriod()` function currently accepts a user-supplied `_initialPrice`:

```
function startTwapPeriod(uint256 _initialPrice) external onlyOwner {
    if (lastPriceUpdateTimestamp != 0) {
        revert AltoRewardsOracleAlreadyStartedTwapPeriod();
    }

    lastPriceUpdateTimestamp = block.timestamp;
    cachedPrice = _initialPrice;

    emit PriceUpdated(block.timestamp, cachedPrice, 0);
}
```

Meanwhile, the `updatePrice()` function demonstrates that the oracle can fetch the current price programmatically:

```
function updatePrice() external {
    // ... validation checks ...

    // reads the amount of out currency for 1 full unit of in-currency
    uint256 newPrice = readUniswapPool(inTokenOneUnit, uniswapNode);

    // converts the price from the out-currency to chainlinkNode.outDecimals decimals
    cachedPrice = readChainlinkFeed(newPrice, chainlinkNode);

    lastPriceUpdateTimestamp = block.timestamp;

    emit PriceUpdated(block.timestamp, newPrice, oldPrice);
}
```

Requiring manual input of the initial price creates an opportunity for human error. If the owner provides an incorrect initial price, it could affect the TWAP calculation used in the AltoRewardsDistributor for pricing call options.

Recommendation: Consider refactoring the price fetching logic into an internal function that can be shared between `startTwapPeriod()` and `updatePrice()`:

```
function startTwapPeriod() external onlyOwner {
    if (lastPriceUpdateTimestamp != 0) {
        revert AltoRewardsOracleAlreadyStartedTwapPeriod();
    }

    _updatePrice();
}

function updatePrice() external {
    if (lastPriceUpdateTimestamp == 0) {
        revert AltoRewardsOracleNotInitiated();
    }

    if (block.timestamp - lastPriceUpdateTimestamp <= priceUpdateCooldownPeriod) {
        revert AltoRewardsOracleTooEarlyToUpdatePrice();
    }

    _updatePrice();
}
```

```
function _updatePrice() internal {
    uint256 oldPrice = cachedPrice;

    // reads the amount of out currency for 1 full unit of in-currency
    uint256 newPrice = readUniswapPool(inTokenOneUnit, uniswapNode);

    // converts the price from the out-currency to chainlinkNode.outDecimals decimals
    cachedPrice = readChainlinkFeed(newPrice, chainlinkNode);

    lastPriceUpdateTimestamp = block.timestamp;

    emit PriceUpdated(block.timestamp, newPrice, oldPrice);
}
```

This approach eliminates the need for manual price input, reducing the risk of configuration errors while ensuring consistency in how prices are fetched throughout the contract's lifecycle.

Alto: Acknowledged.

Phaze: Acknowledged.

3.3.3 Missing validation in Uniswap direction configuration could result in inverted price oracle

Severity: Informational

Context: AltoRewardsOracle.sol#L142-L147

Description: The `AltoRewardsOracle` contract lacks validation to ensure that the Uniswap pool direction is configured correctly. This could result in the oracle returning inverted prices (e.g., USDC/ALTO instead of ALTO/USDC), which would break the pricing mechanism in the `AltoRewardsDistributor` contract where users exercise call options to purchase ALTO tokens.

The `_updateInTokenOneUnitFromUniswapNode()` function determines which token from the Uniswap pool is the "in token" (ALTO) based on the configured direction:

```
function _updateInTokenOneUnitFromUniswapNode() internal {
    address inToken = uniswapNode.direction == ConversionDirection.IN_TO_OUT
        ? IUniswapV3Pool(uniswapNode.pool).token0()
        : IUniswapV3Pool(uniswapNode.pool).token1();
    inTokenOneUnit = 10 ** IERC20Metadata(inToken).decimals();
}
```

This function is called during initialization and whenever the Uniswap node configuration is updated. However, there is no validation to ensure that the determined `inToken` is actually the ALTO token that the oracle is intended to price.

If the direction is configured incorrectly, the oracle will:

1. Use the wrong token's decimals to calculate `inTokenOneUnit`.
2. Request a price quote for the wrong token from the Uniswap pool.
3. Return an inverted price (e.g., how much ALTO you can buy for 1 USDC instead of how much USDC you need to buy 1 ALTO).

The oracle's `getPrice()` function is used by the `AltoRewardsDistributor` to calculate payment amounts:

```
// In AltoRewardsDistributor._calculatePaymentAmount()
uint256 rewardPriceValue = IUsdOracle(rewardTokenUsdOracle).getPrice();
// ... price calculation logic ...
```

An inverted price would cause users to either pay drastically too much or too little when exercising their call options, depending on which direction the inversion occurs.

Recommendation: Add validation in `_updateInTokenOneUnitFromUniswapNode()` to ensure the determined token matches the expected ALTO token address:

```
function _updateInTokenOneUnitFromUniswapNode() internal {
    address token0 = IUniswapV3Pool(uniswapNode.pool).token0();
    address token1 = IUniswapV3Pool(uniswapNode.pool).token1();

    address inToken = uniswapNode.direction == ConversionDirection.IN_TO_OUT ? token0 :
    ↪  token1;
    address outToken = uniswapNode.direction == ConversionDirection.IN_TO_OUT ? token1 :
    ↪  token0;

    // Validate that inToken appears to be ALTO and outToken appears to be a stablecoin
    // This could check token symbols, known addresses, or other heuristics
    if (!_isAltoToken(inToken) || !_isEther(outToken)) {
        revert AltoRewardsOracleInvalidDirection();
    }

    inTokenOneUnit = 10 ** IERC20Metadata(inToken).decimals();
}
```

Alto: Fixed in PR 407.

Phaze: Fix verified.

3.3.4 Mutable oracle configurations increase risk of misconfiguration

Severity: Informational

Context: AltoRewardsOracle.sol#L124-L147

Description: The oracle contracts expose owner-controlled functions that allow updating critical configuration parameters after deployment. Both `AltoMultiChainlinkLendingOracle` and `AltoRewardsOracle` include functions to modify their Chainlink and Uniswap data sources:

- In AltoMultiChainlinkLendingOracle:

```
function updateChainlinkNodes(ChainlinkNode[] memory _chainlinkNodes) external
↪  onlyOwner {
    if (_chainlinkNodes.length == 0) {
        revert AltoMultiChainlinkLendingOracleInvalidInput();
    }
    emit SetChainlinkNodes(chainlinkNodes, _chainlinkNodes);
    delete chainlinkNodes;
    for (uint256 i = 0; i < _chainlinkNodes.length; i++) {
    _validateChainlinkParams(_chainlinkNodes[i]);
    chainlinkNodes.push(_chainlinkNodes[i]);
    }
}
```

- In AltoRewardsOracle:

```
function updateChainlinkNode(ChainlinkNode memory _chainlinkNode) external
↪  onlyOwner {
    _validateChainlinkParams(_chainlinkNode);
    emit SetChainlinkNode(chainlinkNode, _chainlinkNode);
    chainlinkNode = _chainlinkNode;
}

function updateUniswapNode(UniswapNode memory _uniswapNode) external onlyOwner {
    _validateUniswapParams(_uniswapNode);
    emit SetUniswapNode(uniswapNode, _uniswapNode);
    uniswapNode = _uniswapNode;
    _updateInTokenOneUnitFromUniswapNode();
}
```

While these functions include parameter validation, they allow the owner to fundamentally change the oracle's data sources at any time. This creates several considerations:

Configuration Complexity: Oracle configurations are referenced in multiple places (the oracle contracts themselves, consumer contracts like `AltoRewardsDistributor`, and deployment scripts), making it harder to track what data sources are actually being used.

Immediate Price Changes: Changing oracle configurations causes immediate price updates. Even with proper validation, these changes happen atomically and any downstream contracts consuming the oracle will immediately see different prices.

Multi-Contract Coordination: If multiple oracles need coordinated updates, there's no atomic way to update them all simultaneously, potentially leaving the protocol in an inconsistent state between updates.

Recommendation: Consider removing update functions from oracle contracts and handling configuration changes by deploying new oracle contracts and updating references at the consumer contract level, e.g. in AltoRewardsDistributor or similar consumer contracts. This approach provides a single source of configuration in consumer contracts, makes oracle changes more explicit through new deployments, and allows new oracles to be tested before activation.

Alto: Acknowledged.

Phaze: Acknowledged.

3.3.5 Uniswap oracle rounds down when calculating prices used for user payments

Severity: Informational

Context: OracleMath.sol#L23-L33

Description: The `_getQuoteAtTick()` function in `OracleMath.sol` uses floor division when calculating price quotes from Uniswap tick data. When these prices are used to determine payment amounts that users must transfer to the protocol (as in `AltoRewardsDistributor.claimRewardTokens()`), rounding down results in users paying slightly less than they should.

The function currently implements:

```
function _getQuoteAtTick(int24 tick, uint256 baseAmount, ConversionDirection direction)
    internal
    pure
    returns (uint256 quoteAmount)
{
    uint256 ratio = _getRatioAtTick(tick);

    quoteAmount = (direction == ConversionDirection.IN_TO_OUT)
        ? _mulDiv(ratio, baseAmount, 1e18)
        : _mulDiv(1e18, baseAmount, ratio);
}
```

The `_mulDiv()` function from `FullMath` performs floor division, which rounds down to the nearest integer. This is called through the following chain:

1. `AltoRewardsDistributor.claimRewardTokens()` calls `_calculatePaymentAmount()`.
2. `_calculatePaymentAmount()` retrieves the ALTO token price via `IUsdOracle(rewardTokenUsdOracle).getPrice()`.
3. `AltoRewardsOracle.getPrice()` calls `readUniswapPool(inTokenOneUnit, uniswapNode)`.
4. `readUniswapPool()` calls `_getQuoteAtTick()` with the time-weighted average tick.

When the ALTO price is calculated with floor division, it returns a slightly lower value than the true price. Subsequently, when `_calculatePaymentAmount()` computes how much users need to pay, this undervalued price results in a lower payment amount.

While `_calculatePaymentAmount()` does round up in its own calculations to avoid zero payments, this doesn't compensate for the initial rounding down in the oracle price itself:

```
// In _calculatePaymentAmount()
uint256 rewardPriceValue = IUsdOracle(rewardTokenUsdOracle).getPrice(); // Already
↪  rounded down
uint256 tokenPrice = IUsdOracle(paymentTokenConfig.oracle).getPrice();
```

```
uint256 rewardToPurchaseCostUsd = rewardsPurchaseAmount.divideWithRounding(
    rewardPriceValue,  // Uses the already-rounded-down price
    10 ** REWARD_TOKEN.decimals,
    true // round up to avoid zero payment
);
```

In DeFi protocols, the standard practice is to round in favor of the protocol when users are making payments (round up) and in favor of users when they are receiving funds (round down). While individual rounding losses are small (typically 1 wei or a few wei per transaction), they can be exacerbated under certain circumstances.

Recommendation: Consider modifying the oracle calculation to support rounding up when the price will be used for user payments. This could be implemented by:

1. Add a rounding parameter to `_getQuoteAtTick()`:

```
function _getQuoteAtTick(
    int24 tick,
    uint256 baseAmount,
    ConversionDirection direction,
    bool roundUp
)
    internal
    pure
    returns (uint256 quoteAmount)
{
    uint256 ratio = _getRatioAtTick(tick);

    if (direction == ConversionDirection.IN_TO_OUT) {
        quoteAmount = roundUp
            ? _mulDivRoundingUp(ratio, baseAmount, 1e18)
            : _mulDiv(ratio, baseAmount, 1e18);
    } else {
        quoteAmount = roundUp
            ? _mulDivRoundingUp(1e18, baseAmount, ratio)
            : _mulDiv(1e18, baseAmount, ratio);
    }
}

function _mulDivRoundingUp(uint256 a, uint256 b, uint256 denominator)
    internal
    pure
    returns (uint256 result)
{
    result = _mulDiv(a, b, denominator);
    if (mulmod(a, b, denominator) > 0) {
        result++;
    }
}
```

2. Round up consistently in oracle operations:

Since the oracle is primarily used for pricing assets in the rewards distributor (where users pay), consider always rounding up in `_getQuoteAtTick()`:

```
function _getQuoteAtTick(int24 tick, uint256 baseAmount, ConversionDirection direction)
    internal
    pure
    returns (uint256 quoteAmount)
{
    uint256 ratio = _getRatioAtTick(tick);

    if (direction == ConversionDirection.IN_TO_OUT) {
        quoteAmount = _mulDiv(ratio, baseAmount, 1e18);
        if (mulmod(ratio, baseAmount, 1e18) > 0) {
            quoteAmount++;
```

```
        }
    } else {
        quoteAmount = _mulDiv(1e18, baseAmount, ratio);
        if (mulmod(1e18, baseAmount, ratio) > 0) {
            quoteAmount++;
        }
    }
}
```

The first option provides more flexibility if the oracle is used in different contexts with different rounding requirements, while the second option is simpler if the oracle's primary use case is for pricing in payment calculations.

Alto: Fixed in PR 409.

Phaze: Fix verified.

3.3.6 Protocol assumptions about ratio growth and rounding directions could be documented explicitly

Severity: Informational

Context: ModuleERC4626Ratio.sol#L30-L36

Description: The protocol makes important assumptions about the behavior of underlying assets and rounding directions that are not explicitly documented in the code. These implicit assumptions affect system safety and correctness, and their absence could lead to integration errors or protocol misuse.

- Assumption 1: Monotonically Increasing Ratios.

  The ERC4626 ratio capping mechanism in `ModuleERC4626Ratio.sol` assumes that wrapped asset ratios (like wstETH/stETH, rETH/ETH, sUSDe/USDe) always increase over time and never decrease:

  ```
  function readLatestRatio(uint256 latestRatio, ERC4626RatioNode memory
  ↪   _erc4626RatioNode)
      public
      view
      returns (uint256)
  {
      uint256 currentRatio = latestRatio;
      if (currentRatio == 0) return 0;

      uint256 maxRatio = _computeMaxRatio(_erc4626RatioNode);

      if (currentRatio > maxRatio) {
          return maxRatio;  // Caps growth but doesn't enforce a minimum
      } else {
          return currentRatio;
      }
  }
  ```

  The function caps the maximum ratio growth to prevent manipulation, but it does not enforce a minimum ratio. This design implicitly assumes that ratios never decrease because:

  - For liquid staking tokens (wstETH, rETH, mETH), the ratio represents accumulated staking rewards, which only grows.
  - For yield-bearing tokens (sUSDe), the ratio represents accrued interest, which only increases.
  - Decreasing ratios would indicate slashing events or protocol losses.

  However, this assumption is nowhere documented in the code or comments.

- Assumption 2: Protocol-Favorable Rounding Directions.

  The protocol makes different rounding decisions based on whether users are depositing or borrowing, but these decisions are not consistently documented:

  In lending/collateral scenarios (protocol should round down):

- Users deposit collateral and borrow against it.
- Oracle prices should round down to avoid overvaluing collateral.
- This keeps the protocol solvent by being conservative about borrowing capacity.
- Example: `ModuleERC4626Ratio.readLatestRatio()` returns capped values that prevent over-valuation.

In payment scenarios (protocol should round up):

- Users pay to acquire protocol tokens (like in AltoRewardsDistributor).
- Prices should round up to ensure users pay enough.
- This prevents value leakage from accumulated rounding losses.
- Current behavior: The Uniswap oracle rounds down, which is incorrect for this use case.

Assumption 3: Directional Usage of Collateral.

The protocol assumes users only:

1. Deposit collateral → Borrow DUSD (not the reverse).
2. Pay DUSD/stablecoins → Acquire ALTO tokens (not the reverse).

If the protocol were to support the opposite flow (depositing DUSD to borrow underlying collateral), the rounding directions would need to be inverted. For example:

- If users could deposit DUSD and borrow wstETH, the oracle should round up the collateral value to prevent under-collateralization.
- If users could sell ALTO tokens for stablecoins, the oracle should round down the ALTO price to protect the protocol.

The current implementation doesn't document these usage patterns, making it easy for future integrators to misuse the oracle functions with incorrect rounding.

Recommendation: Add thorough documentation to make protocol assumptions explicit:

1. Document ratio monotonicity assumptions:

```
/// @title Module ERC4626 Ratio
/// @notice Utility contract used to read the ratio of ERC4626 vaults and LSTs with
↪  growth caps
/// @dev IMPORTANT ASSUMPTION: This module assumes that all underlying asset ratios are
/// monotonically increasing (never decrease). This holds true for:
/// - Liquid staking tokens (wstETH, rETH, mETH) where ratios represent accumulated
↪  rewards
/// - Yield-bearing tokens (sUSDe) where ratios represent accrued interest
/// @dev If an asset experiences slashing or losses causing the ratio to decrease, this
/// module will report the decreased value without safeguards. Such assets should either:
/// 1. Not be integrated into the protocol, or
/// 2. Use a different oracle mechanism that includes minimum ratio enforcement
contract ModuleERC4626Ratio {
    // ...

    /// @notice Returns the current capped ratio
    /// @dev ROUNDING: Always returns floor value (rounds down) to avoid overvaluing
    ↪  collateral
    /// @dev This is appropriate for lending scenarios where users borrow against this
    ↪  collateral
    function readLatestRatio(uint256 latestRatio, ERC4626RatioNode memory
    ↪  _erc4626RatioNode)
        public
        view
        returns (uint256)
    {
        // ...
    }
}
```

2. Document rounding directions:

```
/// @notice Gets a quote for an amount of input currency and returns the output currency
/// @param quoteAmount The input currency amount to get the price of in output currency
/// @param config Uniswap configuration
/// @return The value of the `quoteAmount` in output currency
/// @dev ROUNDING: This function uses floor division (rounds down)
/// @dev APPROPRIATE USES:
///      - Pricing collateral in lending protocols (conservative valuation)
///      - Calculating amounts when USERS RECEIVE tokens
/// @dev INAPPROPRIATE USES:
///      - Pricing assets when users make payments (use ceiling division instead)
///      - Any scenario where protocol should receive more rather than less
function readUniswapPool(uint256 quoteAmount, UniswapNode memory config)
    public
    view
    returns (uint256)
{
    // ...
}
```

3. Add usage guidelines:

```
/// @title Alto Protocol Oracle Usage Guidelines
/// @notice This document describes the correct usage of oracle functions based on
↪  rounding requirements
///
/// LENDING/COLLATERAL SCENARIOS (round DOWN):
/// - When valuing user collateral deposits
/// - When calculating maximum borrowing capacity
/// - Goal: Be conservative to keep protocol solvent
/// - Use: AltoMultiChainlinkLendingOracle, ModuleERC4626Ratio (as-is)
///
/// PAYMENT SCENARIOS (round UP):
/// - When calculating payment amounts users must transfer
/// - When pricing tokens users are purchasing
/// - Goal: Prevent value leakage from accumulated rounding
/// - Use: Modify oracle to support ceiling division, or round up at consumer level
///
/// REVERSE FLOWS (not currently supported):
/// - If protocol adds: deposit DUSD → borrow collateral
///   Then: Oracle must round UP collateral value
/// - If protocol adds: sell ALTO → receive stablecoins
///   Then: Oracle must round DOWN ALTO price
```

4. Add invariant documentation:

```
/// @notice Validates and stores the ERC4626 ratio configuration
/// @dev INVARIANT: initialRatio must be ≤ current on-chain ratio
/// @dev INVARIANT: Over time, ratio should only increase (monotonicity assumption)
/// @dev SECURITY: If ratio decreases, it indicates asset losses/slashing - should
↪  trigger alerts
function _validateERC4626RatioParams(ERC4626RatioNode memory _erc4626RatioNode) internal
↪  view {
    // ...
}
```

This documentation would make the protocol's assumptions explicit, helping auditors verify correctness and preventing future integrators from misusing oracle functions with incorrect rounding directions.

Alto: Fixed in PR 410.

Phaze: Fix verified.