Security Audit Report

Astroport PCL Neutron Duality Orderbook Integration

v1.0

March 17, 2025

Table of Contents

Table of Contents **2**
License **3**
Disclaimer **4**
Introduction **5**
Purpose of This Report 5
Codebase Submitted for the Audit 5
Methodology 6
Functionality Overview 6
How to Read This Report **7**
Code Quality Criteria **8**
Summary of Findings **9**
Detailed Findings **10**
1. Missing ob_state persistence causes incorrect pool repeg 10
2. Updated price state is not persisted after cumulative trade processing 10
3. Inconsistent parameter ordering in price conversion function 11
4. Excessive gas consumption in pool operations 12
5. Disabling of Neutron order book causes pool operations failure 13
6. Incorrect percentage constants in liquidity definitions 14
7. Inconsistent range validation 14
8. Missing validation for avg_price_adjustment parameter 15
9. Missing empty pool validation in query_cumulative_prices 15
10. Inaccurate simulations due to missing cumulative trade handling 16
11. Missing check for negative order prices 16
12. Use of magic numbers decreases maintainability 17
13. Commented-out code 17
Appendix: Test Cases **17**
1. Test case for “Excessive gas consumption in pool operations” 18
2. Test case for “Inconsistent range validation” 19

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by the Neutron Audit Sponsorship Program to perform a security audit of Astroport PCL Neutron Duality Orderbook Integration.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/astroport-fi/astroport-core
Commit	`8f76fb7bc172bc71e04ef666d5da805b6e6c1305`
Scope	The scope is restricted to changes introduced since commit `622f7476f72968ee758e1da2ea6b5391d18934e5`.
Fixes verified at commit	`291aeb693dd559949a16d985bde5a9daf099a80f` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Astroport implements an automated, decentralized exchange protocol in the Cosmos Ecosystem.

This audit exclusively covers Astroport's concentrated liquidity pool integration with Neutron's Duality DEX, located in `contracts/pair_concentrated_duality`.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium	The codebase is well-documented, and the client provided a video call to explain the implementation details.
Test coverage	High	`cargo tarpaulin` reports `72.20%` test coverage.

Summary of Findings

No	Description	Severity	Status
1	Missing `ob_state` persistence causes incorrect pool repeg	Critical	Resolved
2	Updated price state is not persisted after cumulative trade processing	Critical	Resolved
3	Inconsistent parameter ordering in price conversion function	Critical	Resolved
4	Excessive gas consumption in pool operations	Major	Resolved
5	Disabling of Neutron order book causes pool operations failure	Major	Acknowledged
6	Incorrect percentage constants in liquidity definitions	Minor	Resolved
7	Inconsistent range validation	Minor	Resolved
8	Missing validation for `avg_price_adjustment` parameter	Minor	Resolved
9	Missing empty pool validation in `query_cumulative_prices`	Minor	Resolved
10	Inaccurate simulations due to missing cumulative trade handling	Minor	Resolved
11	Missing check for negative order prices	Informational	Resolved
12	Use of magic numbers decreases maintainability	Informational	Resolved
13	Commented-out code	Informational	Resolved

Detailed Findings

1. Missing `ob_state` persistence causes incorrect pool repeg

Severity: Critical

In `contracts/pair_concentrated_duality/src/orderbook/execute.rs:154-218`, the `sync_pool_with_orderbook` function updates the `ob_state` but never saves it.

As a result, critical values such as `pre_reply_balances`, which are used to compute cumulative trades in the `reply` handler, are not stored. This causes the execution to rely on previously set values, which may contain outdated or incorrect data.

Consequently, this can lead to an inaccurate repeg of the pool, affecting pricing and trade execution.

Recommendation

We recommend ensuring that the updated `ob_state` is explicitly before the message execution completes.

Status: Resolved

2. Updated price state is not persisted after cumulative trade processing

Severity: Critical

In `contracts/pair_concentrated_duality/src/reply.rs:42-92`, when handling the `PostLimitOrderCb` reply, the `process_cumulative_trade` function updates the configuration by modifying the price state during trade execution.

However, these modifications are not committed back to storage, leading to potential inconsistencies in the recorded price state across subsequent invocations.

This may result in the loss of price updates, which can impact the accuracy of future trades, liquidity provisions, and other contract operations relying on the correct price scale.

The issue is particularly problematic in a hybrid model that integrates with an order book, where the synchronization between the liquidity pool and order book depends on a consistently updated price state.

Recommendation

We recommend saving the updated configuration back to storage after processing the cumulative trade by adding `CONFIG.save(deps.storage, &config)?;` before returning from the reply handler when handling the `PostLimitOrderCb` case.

Status: Resolved

3. Inconsistent parameter ordering in price conversion function

Severity: Critical

In `contracts/pair_concentrated_duality/src/orderbook/utils.rs:222-233`, within the `SpotOrdersFactory::collect_spot_orders` method, an inconsistency in parameter ordering affects price calculations for sell orders.

The `price_to_duality_notation` function requires a consistent ordering of `base_precision` and `quote_precision` parameters to accurately adjust prices based on precision differences between assets. The function computes the precision difference using the formula:

```
prec_diff = quote_precision as i8 - base_precision as i8
```

However, while the `limit_sell_price` calculation correctly passes parameters as `(precision[0], precision[1])`, the min_average_sell_price calculation erroneously reverses the order to `(precision[1], precision[0])`. This discrepancy leads to an incorrect precision difference computation, potentially causing significant pricing errors in sell orders.

Consequently, miscalculated prices in the order book may lead to unfavorable trades, unexpected behavior, or economic losses for users.

Recommendation

We recommend ensuring parameter ordering consistency when calling `price_to_duality_notation` within the sell order block in `collect_spot_orders`.

Specifically, both `limit_sell_price` and `min_average_sell_price` calculations should use the same parameter ordering.

Status: Resolved

4. Excessive gas consumption in pool operations

Severity: Major

In `contracts/pair_concentrated_duality/src/execute.rs:148`, the `provide_liquidity` function allows users to supply liquidity to the pool.

However, when order book integration is enabled, this operation becomes extremely gas-intensive, potentially leading to out-of-gas failures and block inefficiencies.

This issue extends to all pool operations as they similarly depend on costly order book interactions, making them vulnerable to gas exhaustion and inefficient block execution.

The execution flow for `provide_liquidity` includes, in the worst case where `num_orders` is equal to `ORDER_SIZE_LIMITS.end()`:

1. Performing `60 QuerySimulateCancelLimitOrderRequest` queries to retrieve order states.
2. Processing cumulative trades, updating prices, and computing LP shares.
3. Queuing a mint message.
4. Queuing `60 MsgCancelLimitOrder` and `60 MsgPlaceLimitOrder` messages, and a callback to `ReplyIds::PostLimitOrderCb`.
5. The callback executes `60` additional `QuerySimulateCancelLimitOrderRequest` queries.

Given Neutron's `block.max_gas` limit of `30,000,000`, the gas cost for a single liquidity provision can be estimated as:

- `MsgPlaceLimitOrder` (`good_till_completed`): `31,095` gas
- `MsgCancelLimitOrder`: `25,451` gas
- Total for `60` orders: `3,392,760` gas
- Additional costs: Queries, minting, callbacks, and computation.

Consequently, the theoretical maximum is less than `8.84 provide_liquidity` calls per block.

An empirical test case defined in the Appendix describes the gas used per `provide_liquidity` call equal to `5,096,372`, resulting in a maximum of `5 provide_liquidity` transactions per block.

Since only five transactions fit within a block, the execution order depends on the mempool state. If a transaction is executed while less than `1/5` of block gas remains, it will likely fail due to insufficient gas or be directly discarded by the block proposer. This makes it impractical for

a basic action like providing liquidity to consume such a large portion of the block's resources, especially on a shared blockchain with multiple competing protocols.

This inefficiency applies not only to `provide_liquidity` but to all pool operations as they all depend on canceling and registering orders in the order book.

Recommendation

We recommend reducing the number of messages and interactions with the order book to lower the gas costs of individual pool operations.

As a short-term mitigation, we suggest limiting the maximum number of orders per side, which is currently set at `30`, to a lower value to improve efficiency and prevent excessive resource consumption.

Status: Resolved

5. Disabling of Neutron order book causes pool operations failure

Severity: Major

The Neutron order book is designed to allow governance to disable it, preventing the setting or cancellation of orders as the associated messages will fail and revert.

However, since all pool operations rely on calling the `cancel_orders` function, disabling the Neutron orderbook results in the failure and reversion of all pool operations, rendering the pool non-functional.

This issue also arises if the pool administrator tries to disable the order book integration. Specifically, the `process_custom_msgs` function in `contracts/pair_concentrated_duality/src/execute.rs:693-727` will fail when the orderbook is disabled, as it invokes the `cancel_orders` function, which will revert.

As a result, disabling the Neutron order book effectively locks the pool, even though it could otherwise function using its internal automated market maker logic.

Recommendation

We recommend decoupling pool operations from the orderbook's availability by implementing fallback mechanisms that allow the pool to continue functioning independently when the Neutron orderbook is disabled.

Status: Acknowledged

The client states that, with the current design, they cannot continue operations while a portion of their liquidity remains locked in the order book.

For example, if 50% of the liquidity is allocated to the order book, it would be frozen at outdated prices while the remaining liquidity is utilized in PCL. When the market resumes, these stale prices could disrupt the PCL repegging algorithm, potentially rendering it inoperable.

6. Incorrect percentage constants in liquidity definitions

Severity: Minor

In `contracts/pair_concentrated_duality/src/orderbook/consts.rs:9`, the percentage constants for liquidity are defined as:

- `MIN_LIQUIDITY_PERCENT - 1e16 (0.01%)`
- `MAX_LIQUIDITY_PERCENT - 5e17 (5%)`

However, the `MIN_LIQUIDITY_PERCENT` constant is documented as `0.01%`, but `1e16` as a raw `Decimal` value represents `1%`, not `0.01%`.

This inconsistency could lead to improper validation of liquidity percentage parameters, disallowing values that are between the `0.01%` and `1%` range.

Recommendation

We recommend correcting the raw decimal values to represent the intended percentages. For `0.01%`, the correct value would be `1e14`.

Status: Resolved

7. Inconsistent range validation

Severity: Minor

In `contracts/pair_concentrated_duality/src/orderbook/state.rs:27-39`, the `validate_param` macro uses the range operator `$min..$max`, which is exclusive of the upper bound.

However, the error message associated with this validation states the requirement as "`{min} <= {name} <= {max}`", implying that the upper bound is included.

For instance, if a parameter is intended to be within the inclusive range `[1, 10]`, the macro actually permits values from `1` through `9`, potentially leading to unexpected behavior.

This discrepancy can cause confusion among contract administrators.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend aligning the validation logic with the error message by modifying the range operator to `$min..=$max` to ensure the upper bound is included.

Status: Resolved

8. Missing validation for `avg_price_adjustment` parameter

Severity: Minor

In `contracts/pair_concentrated_duality/src/orderbook/state.rs:69-86`, the `avg_price_adjustment` parameter is imported from the configuration without explicit validation. Unlike `orders_number` and `liquidity_percent`, which undergo thorough validation with defined bounds, `avg_price_adjustment` lacks any programmatic checks.

Specifically, within the `validate` method, there is no verification of the `avg_price_adjustment` value.

This absence of validation poses a risk: if an administrator sets an extreme value for `avg_price_adjustment`, it could result in unfair pricing for users.

Recommendation

We recommend implementing explicit validation for the `avg_price_adjustment` parameter within the `validate` method. This validation should include defined bounds to prevent the parameter from being set to extreme values that could negatively impact pricing fairness.

Status: Resolved

9. Missing empty pool validation in `query_cumulative_prices`

Severity: Minor

In `contracts/pair_concentrated_duality/src/queries.rs:233-260`, the `query_cumulative_prices` function does not validate if pools have sufficient liquidity before performing price calculations, while other query functions like `query_lp_price` properly implement this check.

When pools are empty, this could lead to unexpected behavior, including potential division by zero errors or return of incorrect price data, affecting the reliability of the protocol's price feed information.

Recommendation

We recommend adding empty pool validation at the beginning of the price calculation in `query_cumulative_prices`.

Status: Resolved

10. Inaccurate simulations due to missing cumulative trade handling

Severity: Minor

In `contracts/pair_concentrated_duality/src/queries.rs`, the `query_simulate_provide` (lines `347-379`) and `query_simulation` (lines `136-195`) functions help users estimate liquidity provision and swaps, respectively.

However, both functions fail to account for the cumulative trade repeg that occurs before `calculate_shares` is executed.

As a result, their simulated outputs may differ from actual execution outcomes if unhandled trades exist in the order book.

Recommendation

We recommend updating both functions to incorporate the cumulative trade repeg before performing calculations, ensuring simulations accurately reflect real execution results.

Status: Resolved

11. Missing check for negative order prices

Severity: Informational

In `contracts/pair_concentrated_duality/src/orderbook/utils.rs:153-158`, when constructing sell orders, the price calculation logic for orders after the first one could theoretically produce negative values if the accumulated sell-side liquidity exceeds the expected asset amount.

While there is a check for zero prices, there is no explicit check against negative prices. If `self.orderbook_one_side_liquidity(false)` exceeds `asset_1_sell_amount`, the result would be negative. Even though it is unlikely in normal market conditions, this edge case lacks validation.

Recommendation

We recommend adding an explicit check to handle potential negative values.

Status: Resolved

12. Use of magic numbers decreases maintainability

Severity: Informational

Throughout the codebase, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers are listed below:

- `contracts/pair_concentrated_duality/src/orderbook/utils.rs:273`

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Resolved

13. Commented-out code

Severity: Informational

In `contracts/pair_concentrated_duality/src/orderbook/state.rs:280`, there is a substantial block of commented-out code.

This commented code appears to be an alternative implementation of the current functionality. Maintaining commented-out code increases maintenance burden and can lead to confusion about which implementation is correct or preferred. It also suggests incomplete development or testing.

Recommendation

We recommend either removing the commented-out code if it's no longer needed.

Status: Resolved

Appendix: Test Cases

1. Test case for “Excessive gas consumption in pool operations”

```
#[test]
fn test_gas_usage_provide_liquidity() {
    let test_coins = vec![TestCoin::native("astro"),
TestCoin::native("untrn")];
    let orders_number = 30;

    let app = NeutronTestApp::new();
    let neutron = TestAppWrapper::bootstrap(&app).unwrap();
    let owner = neutron.signer.address();

    let astroport = AstroportHelper::new(
        neutron,
        test_coins.clone(),
        ConcentratedPoolParams {
            price_scale: Decimal::from_ratio(1u8, 2u8),
            ..common_pcl_params()
        },
        OrderbookConfig {
            executor: Some(owner),
            liquidity_percent: Decimal::percent(20),
            orders_number,
            min_asset_0_order_size: Uint128::from(1_000u128),
            min_asset_1_order_size: Uint128::from(1_000u128),
            avg_price_adjustment: Decimal::from_str("0.0001").unwrap(),
        },
    )
    .unwrap();

    astroport.enable_orderbook(&astroport.owner, true).unwrap();

    let user = astroport
        .helper
        .app
        .init_account(&[
            coin(2_000_000_000000u128, "untrn"),
            coin(2_000_000_000000u128, "astro"),
        ])
        .unwrap();

    let initial_balances = [
```

```
astroport.assets[&test_coins[0]].with_balance(10_000_000000u128),

astroport.assets[&test_coins[1]].with_balance(20_000_000000u128),
    ];

    // Providing initial liquidity
    let res = astroport
        .provide_liquidity(&user, &initial_balances)
        .unwrap();


    println!("Gas used for provide liquidity: {}",
res.gas_info.gas_wanted);
    println!("Amount of provide liquidity messages that can fit a block:
{}", 30_000_000 / res.gas_info.gas_wanted);
}
```

2. Test case for "Inconsistent range validation"

```
#[test]
fn test_validate_param_failure() {
    // This should fail because 10 is NOT within the range [1, 10) (10
is excluded)
    let param_value = 10;
    let result: Result<(), StdError> = (|| {
        validate_param!(param_value, param_value, 1, 10);
        Ok(())
    })();
    assert!(result.is_err(), "Expected value 10 to be invalid");

    if let Err(e) = result {
        // The error message indicates the intended inclusive range, but
the macro uses an exclusive range.
        let expected_message = "Generic error: Incorrect orderbook
params: must be 1 <= param_value <= 10, but value is 10";
        assert_eq!(e.to_string(), expected_message);
    }
}
```